Digital Design For Neuromorphic Bio-Inspired Vision Processing

by

Amirreza Yousefzadeh

Submitted to the Department of Physics
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

UNIVERSITY OF SEVILLA

December 2017

Author .....
Amirreza Yousefzadeh
Department of Physics
December 15, 2017

Certified by .....
Bernabe Linares-Barranco
Full Professor of Research
Thesis Supervisor

Certified by .....
Teresa Serrano Gotarredona
Research Scientist
Thesis Supervisor

Certified by .....
Timothee Masquelier
Research Scientist
Thesis Supervisor

Acknowledgments

I owe a debt of gratitude to my advisor Professor Bernabe Linares-Barranco for his patience, vision, and foresight. He wisely guided and encouraged me during the time of research and writing of this thesis. It is my pleasure to acknowledge Professor Teresa Serrano Gotarredona who was a vital benefactor in the completion of this work. I am especially grateful to the Timothee Masquelier and Simon Thorpe from the Brain & Cognition Research Center (CerCo, Toulouse) and Garrick Orchard and his team (Bala-Krishna, Sherine, Sharad, ...) from the SINAPSE (NUS, Singapore), for giving me a very stimulating, intellectual and inspiring experience during my stays in their institutions. I was lucky that I had this chance to work in such an interdisciplinary team. I would like to thank Peter Van Der Made and Anil Mankar from BrainChip company for their support for commercialization of our ideas. I thank the members of the Neuromorphic Engineering Group in Sevilla Microelectronic Institute (IMSE-CNM) for productive meetings, discussions, and support. I appreciate to my friends Laurentiu, Luis, Manuel, Charan and all the other colleagues from IMSE. Lastly, I would like to thank my family for all their love and encouragement. For my parents who raised me with respect for science and supported me in all my pursuits. And most of all for my loving, supportive, encouraging, and patient wife Sahar for faithful support during the final stages of this Ph.D. is so appreciated. The author's work was supported by a four-year FPI Predoctoral scholarship financed by Superior Council of Scientific Investigations (CSIC).

Digital Design For Neuromorphic Bio-Inspired Vision Processing

by

Amirreza Yousefzadeh

Submitted to the Department of Physics
on December 15, 2017, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Neuromorphic engineering is a new emerging technology that tries to bring intelligence and efficiency of biological brain to silicon hardware. Conventional processors are not able to simulate biological neurons efficiently due to the entirely different structure of neural networks. Neuromorphic engineering tries to provide efficient solutions for implementation of bio-inspired processors. To contribute in this exciting field, we focus on three significant challenges related to communication links for transmitting spikes between neurons, development of proper algorithms to process these spikes and design general purpose neuromorphic hardware that can learn like a biological brain.

Thesis Supervisor: Bernabe Linares-Barranco
Title: Full Professor of Research

Thesis Supervisor: Teresa Serrano Gotarredona
Title: Research Scientist

Thesis Supervisor: Timothee Masquelier
Title: Research Scientist

This doctoral thesis has been examined by a Committee as follows:

. . . . . .
.

. . . . . .
.

. . . . . .
.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Artificial Intelligence (AI) is an exciting technology that flourished in this century. One of the goals for this technology is to give learning ability to computers. Currently, machine intelligence surpasses human intelligence in specific domains. Besides some conventional machine learning algorithms, Artificial Neural Networks (ANNs) is arguably the most exciting technology that is used to bring this intelligence to the computer world. Due to ANN's advanced performance, increasing number of applications that need kind of intelligence are using ANN. Neuromorphic engineers are trying to introduce bio-inspired hardware for efficient implementation of neural networks. This hardware should be able to simulate a vast number of neurons in real-time with complex synaptic connectivity while consuming little power.

The work that has been done in this thesis is hardware oriented, so it is necessary for the reader to have a good understanding of the hardware that is used for developments in this thesis. In this chapter, we provide a brief overview of the hardware platforms that are used in this thesis. Afterward, we explain briefly the contributions of this thesis to the bio-inspired processing research line.

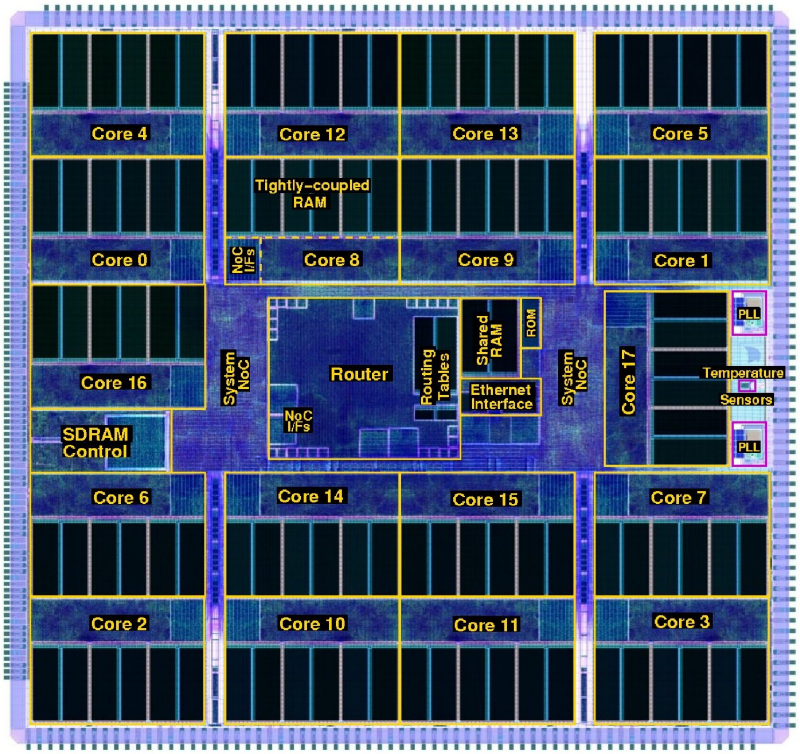


Figure 1-1: SpiNNaker chip layout [1]. It contains 18 ARM processors, a Router and SDRAM controller.

1.1 Hardware Platforms

1.1.1 SpiNNaker neuromorphic platform

SpiNNaker (Spiking Neural Network Architecture) [1] is a massively parallel multi-core system that is optimized for neuromorphic applications. Fig. 1-1 shows the layout of the currently available SpiNNaker chip[1]. SpiNNaker chip contains 18 ARM968 processors[2] each with 64kB of tightly-coupled data memory and 32kB of tightly-coupled instruction memory. The chip contains a Globally Asynchronous Locally Synchronous (GALS) architecture with an asynchronous packet switching network that is highly optimized for neuromorphic applications [10].

ARM cores in SpiNNaker chip communicate to each other and to outside through

[1]SpiNNaker2 is currently under development and in this thesis, we only used the first version of SpiNNaker chip

[2]One ARM processor is used for management, and another ARM core is reserved. Only 16 ARM cores are involved in the neuromorphic process

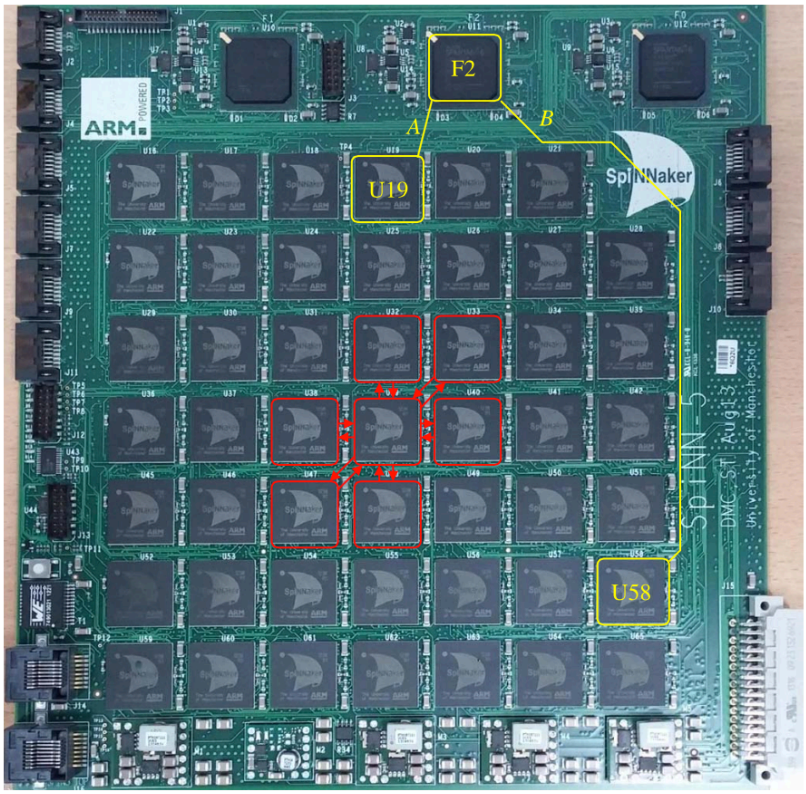


Figure 1-2: SPIN-5 board that contains 48 SpiNNaker chips each with 18 ARM processors (864 ARM processors in total). Each SpiNNaker chip is connected to 6 chips in its neighborhood as it is shown with red color. The chips in the border are connected to FPGAs as it is shown with yellow color.

the router. The router has 24 asynchronous bidirectional links, 18 of them are connected to the 18 ARM processors and six of them are connected to external links. Each processor can host several neurons and send/receive spikes to other processors through the router and packet switch network [11]. In this thesis, we used the SPIN-5 board that is shown in Fig. 1-2. This board contains 48 SpiNNaker chips and 3 Spartan-6 FPGAs for communication to other boards. This board is used to build a million core massively parallel computer for human brain simulation [12].

1.1.2 Dynamic Vision Sensor (DVS)

Dynamic Vision Sensors generate and transmit information of the visual scene in a completely different method than conventional frame-based cameras. Frame-based cameras were invented initially to take static photos. Later-on these cameras were

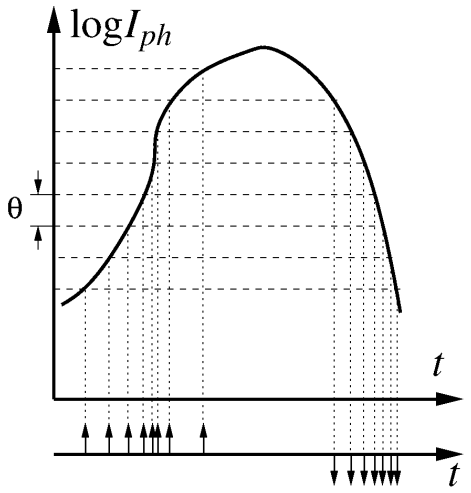
used to capture videos by taking fast and consecutive images at regular time intervals. This method of video capturing is inefficient when dynamics of the visual scene is faster or slower than the frame rate. Dynamic Vision Sensors (DVSs) contain independent asynchronous pixels that generate events when they detect a change in light intensity. These type of vision sensors are a subtype of the so-called silicon retina because their feature is inspired by biological retinas. Dynamic Vision Sensors have several advantages that result in their growing use in neuromorphic engineering and low power mobile applications.

Interest in event-driven vision sensors has increased rapidly in recent years as the technology has matured, become more accessible to researchers, and as the sensors' potential to enable low-latency sensing at low computational cost has become clear. However, event-driven vision sensors operate differently than more traditional frame-based vision sensors and therefore processing event data requires a different approach than frame-based data.

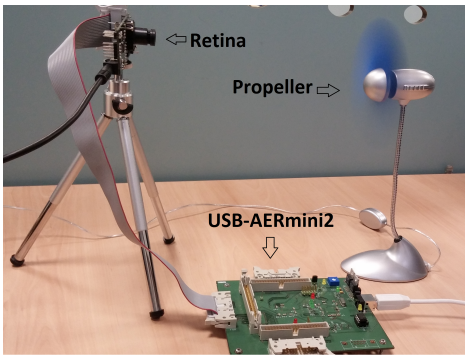
Dynamic Vision Sensors output data in the Address Event Representation (AER) format [13], where each event consists of a pixel address and a single bit. The single bit indicates whether the intensity change was positive or negative. The concept is shown in Fig. 1-3. As shown in Fig. 1-3(a), when logarithmic change of light intensity passes a threshold (θ), an event will be generated. Fig. 1-3(c) shows the reconstruction of DVS events for a moment when a propeller is rotating in front of the DVS. The recording setup is shown in Fig. 1-3(b). Reconstruction is done with jAER software [4] where black dots indicate events with negative polarity and white dots indicate events with positive polarity.

Dynamic Vision Sensors can outperform frame-based vision sensors regarding data compression[3], dynamic range, temporal resolution and power efficiency. Various different event-based temporal contrast vision sensors exist [14]. Most recently Son et al. [15] from Samsung presented a 640×480 pixel DVS which consumes a total of 27mW at the data rate of 100keps and 50mW at 300Meps. This chip has better temporal

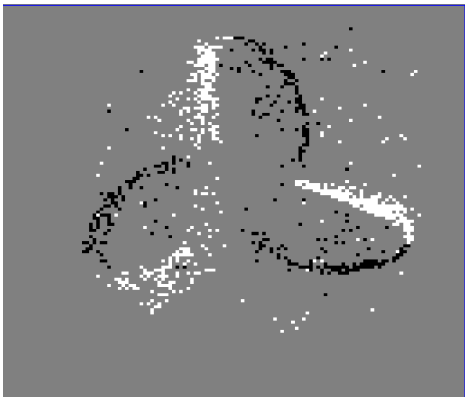
[3]While frame-based cameras generate redundant frames from static scene, DVS only generates events when a movement happens in the scene



(a)



(b)



(c)

Figure 1-3: (a) Event generation for a pixel of DVS when changing in light intensity passes a pre-defined threshold (θ). For positive/negative change, a positive/negative event will be generated. (b) Propeller rotating in front of DVS [2], USB-AERmini2 board [3] time-stamps events from DVS and sends them to computer through USB. (c) Reconstructed output of DVS with jAER software [4]. It contains 864 events within 624us (1.3M events per second)

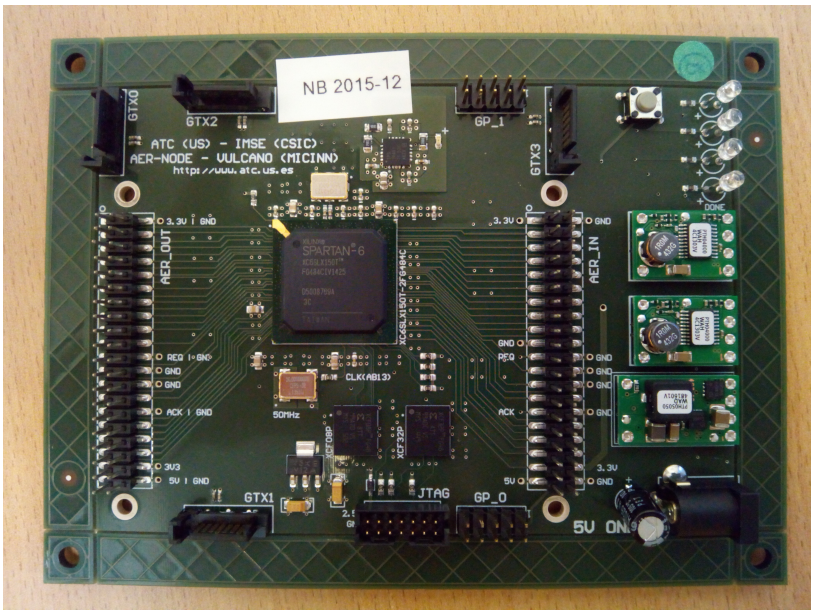


Figure 1-4: AER-NODE board

resolution than a 2000fps camera and wide dynamic range of more than 80db.

1.1.3 AER-NODE board

Fig. 1-4 shows an AER-NODE board [6]. This board contains a SPARTAN-6 FPGA from Xilinx (XC6SLX150T-3), two parallel AER ports (as input and output using the CAVIAR connector [3]) and four bidirectional high-speed serial LVDS links.

1.1.4 USB-AER, jAER and Event logger/player

To send/receive AER events to/from conventional computers, a board that is called USB-AERmini2 (or briefly USB-AER in this thesis) is used. This board is shown in Fig. 1-5. This board supports bi-directional communication with a computer through the USB port. When it receives events from a parallel AER port, it adds time-stamps on the AER events and sends them to the computer through USB port. On the reverse direction, the USB-AERmini2 board receives AER events with time-stamp from the computer, stores them in a buffer and sends them to the parallel AER output port in the proper time.

Inside the computer, a software called jAER (Java-AER) [4] communicates with

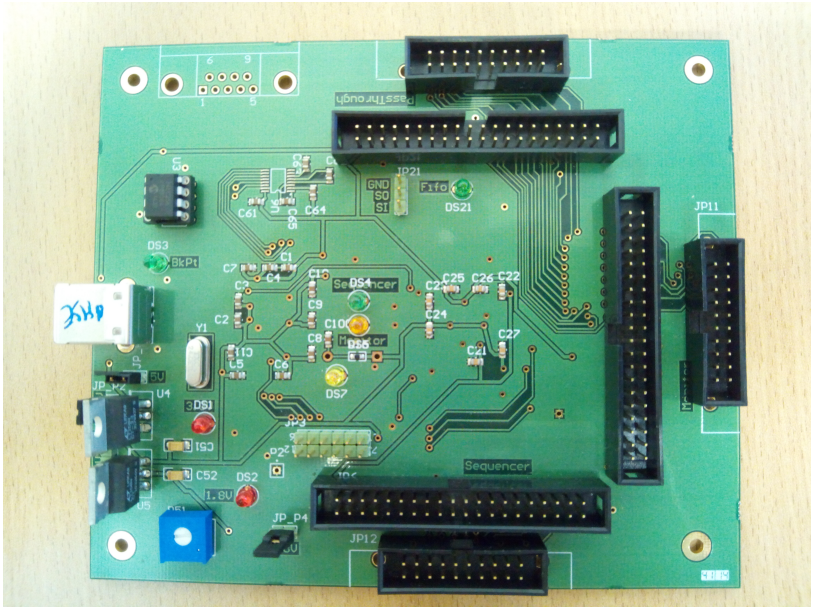


Figure 1-5: USB-AERmini2 board contains one USB port to communicate with computer and three parallel AER ports.

the USB-AERmini2 board. This software takes the AER events flowing into the computer and records them in a file. On the other hand, this software can read AER events from a recorded file and sequence them back toward the USB-AERmini2 board. The jAER software also builds frames using various methods (fixed frame time, fixed number of events) and render them on the computer screen for convenient visualization. It can also perform additional processing (filtering), and users can add additional functionalities as it is an open source project. Fig. 1-3 (c) shows a frame from reconstructed DVS events in jAER software.

Another board called USBAER (or Event Logger/Player or data player in this thesis) is used in this thesis. To benchmark performance of a bio-inspired processing system, we need to use pre-recorded data-sets. This board records events in its 512MB of memory and then plays them back for real-time processing and benchmarking. Since this board stores data locally, the play back is more precise in time and time precision can be as low as 20ns. Fig.1-6 shows a USBAER board.

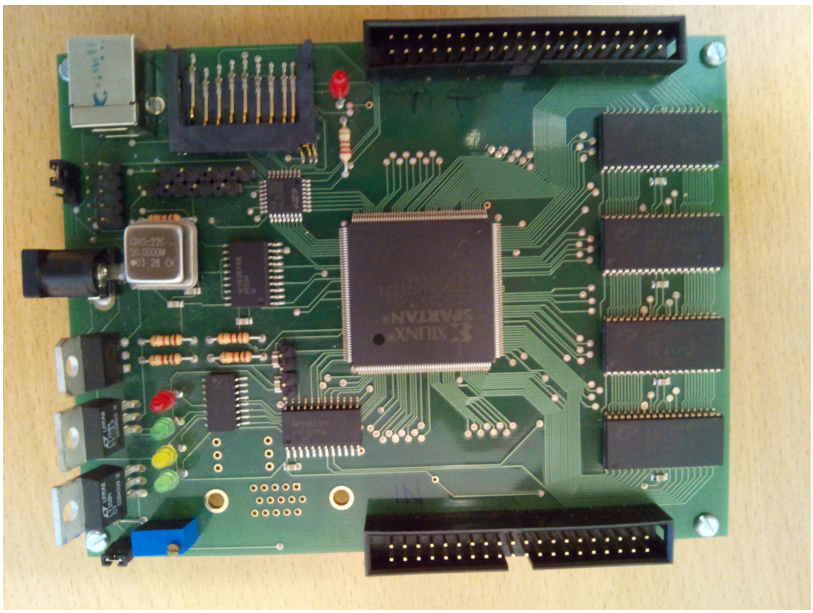


Figure 1-6: USBAER board to record and play back AER events

1.2 Contributions in Bio-inspired processing

We have tried to improve existing solutions for three significant challenges in neuromorphic systems that will be explained here in brief.

The first challenge was related to communications. A simple bio-inspired spiking neural network can contain thousands to millions of neurons in a single chip or array of chips. These neurons need to communicate to each other through a sophisticated and dynamic network of synapses. Asynchronous communication among massively parallel neuromorphic chips can facilitate scalability. In this thesis, we proposed two event-driven asynchronous communication protocols to remove communication bottle-necks of existing neuromorphic hardware. Chapter 2 describes a new method to increase throughput of existing asynchronous links when at least one side of the communication is locally synchronous. Chapter 3 describes a scalable multi-channel fast gigabit serial link that is proposed for neuromorphic applications.

The second challenge was related to the development of hardware implementations for real-time bio-inspired vision processing. We have used the bio-inspired Dynamic Vision Sensor (DVS) as input sensor. The DVS output signals need special signal processing algorithms to extract meaningful information, as well as distinctive hard-

ware design to implement the algorithms efficiently. In Chapter 4 we have introduced an efficient digital implementation for convolutional spiking neural network on FPGAs to process output events from DVS. In Chapter5 we have introduced a Hybrid Neural Network that can be trained by using conventional frame-based techniques while using DVS as its input source. Additionally, similar to biological retina, a DVS needs motion to generate output spikes. In chapter 6 we introduce a robotic platform along with an efficient algorithm to perform micro-saccades with a DVS for object recognition. In chapter 7 we have presented a modification of the current SpiNNaker operating system for real-time event-driven applications and compare it with the conventional time-step-driven operating system.

The last challenge is about designing an efficient neuromorphic hardware with online learning. We have proposed a neuromorphic digital architecture that can learn different types of meaningful stimuli with unsupervised STDP mechanism. Chapter 8 describes our Convolutional STDP core for FPGAs, which can learn visual patterns without supervision. This core also contains a layer of supervised STDP neurons for classification.

Chapter 2

Fast Predictive Handshaking in Synchronous FPGAs for Fully Asynchronous Multisymbol Chip Links: Application to SpiNNaker 2-of-7 Links

This work has been published in:

A. Yousefzadeh, L. A. Plana, S. Temple, T. Serrano-Gotarredona, S. B. Furber and B. Linares-Barranco, "Fast Predictive Handshaking in Synchronous FPGAs for Fully Asynchronous Multisymbol Chip Links: Application to SpiNNaker 2-of-7 Links," in IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 63, no. 8, pp. 763-767, Aug. 2016.

2.1 Abstract

Asynchronous handshaken inter-chip links are very popular among neuromorphic full-custom chips due to their delay-insensitive and high-speed properties. Of special interest are those links that minimize bit-line transitions for power saving, such

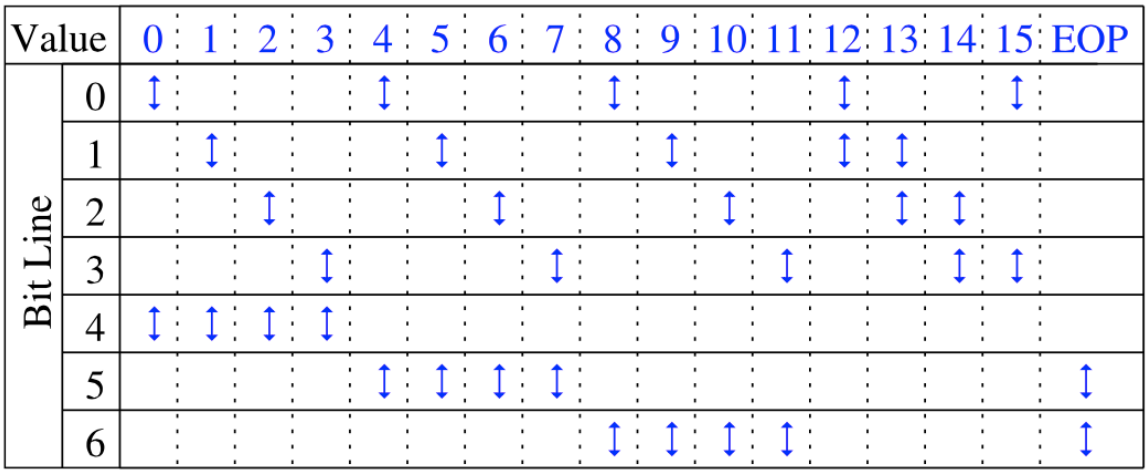


Figure 2-1: Encoding of 2-of-7 NRZ protocol transitions to 4-bit symbol values.

as the two-phase handshaken non-return-to-zero (NRZ) 2-of-7 protocol used in the SpiNNaker chips. Interfacing such custom chip links to field-programmable gate arrays (FPGAs) is always of great interest, so that additional functionalities can be experimented and exploited for producing more versatile systems. Present-day commercial FPGAs operate typically in synchronous mode, thus making it necessary to incorporate synchronizers when interfacing with asynchronous chips. This introduces extra latencies and precludes pipelining, deteriorating transmission speed, particularly when sending multi-symbols per unit communication packet. In this chapter, we present a technique that learns to estimate the delay of a symbol transaction, thus allowing a fast pipelining from symbol to symbol. The technique has been tested on links between FPGAs and SpiNNaker chips, achieving the same throughput as fully asynchronous synchronizerless links between SpiNNaker chips. The links have been tested for periods of over one week without any transaction failure. Verilog codes of FPGA circuits are available freely for academic purpose.

2.2 Introduction

Neuromorphic chips and systems use typically the asynchronous four-phase handshaken address event representation (AER) scheme to interchange information in an event-driven manner [16] [17] for vision systems [18], [19] and robotics [20]. The

recently available multi-ARM-core SpiNNaker chips [1] (intended for simulating large-scale neuromorphic systems) use a special multi-symbol very low-power two-phase-handshaking non-return-to-zero (NRZ) protocol [21], which is called 2-of-7 [10]. Each link is unidirectional and uses eight lines (seven for data and one for Ack, i.e., Acknowledge). A symbol is transmitted by changing the state of two data lines only, which signals a Request for the handshaking. Although there are 21 possible transitions in two lines out of seven lines, only 17 are used by SpiNNaker (16 data symbols and one "End-of-Packet" symbol). This way, data symbols can be represented by 4-bit nibbles, as illustrated in Fig. 2-1. SpiNNaker chips can communicate a packet (also called "event") of either short format (44-bits) with 11 4-bit symbols or long format (76-bits) with 19 4-bit symbols.

The structure of a packet/event is 8-bit header, 32-bit data, 32-bit optional payload (extra data for the long format), and "End-of-Packet" symbol. Fig. 2-2 shows a commercial SpiNN5 board hosting 48 SpiNNaker chips. Each chip connects to six neighbor chips (north, south, east, west, northeast, and southwest), emulating a hexagonal grid [6]. Each chip-to-chip connection contains a pair of 8-bit 2-of-7 lines, one for each direction. Interchip links (which need minimum PCB trace length) can exchange short-format events at a rate of about 6 Meps (mega events per second), which accounts to about 15 ns per symbol transaction. On the top of the board, one can see three Spartan6 field-programmable gate arrays (FPGAs). They connect to some of the SpiNNaker chips through a bidirectional pair of 8-bit 2-of-7 NRZ asynchronous links. On the top in Fig. 2-2, we highlight two of such links: link A between the central FPGA F2 and the SpiNNaker chip U19 on the top row, and link B between the same FPGA and the chip U58 far away and close to the bottom right edge. The circuitry inside the FPGA is clocked and requires the use of synchronizers to interface properly with external asynchronous links [22], [23]. In the next section, we briefly describe how such a standard link would operate, achieving a maximum average throughput of 3.51 Meps from the FPGA to the SpiNNaker chip. Afterwards, we present the new proposed approach, which can reach up to 6.89 Meps from the FPGA to the SpiNNaker chip. Our proposed approach could not be used to improve

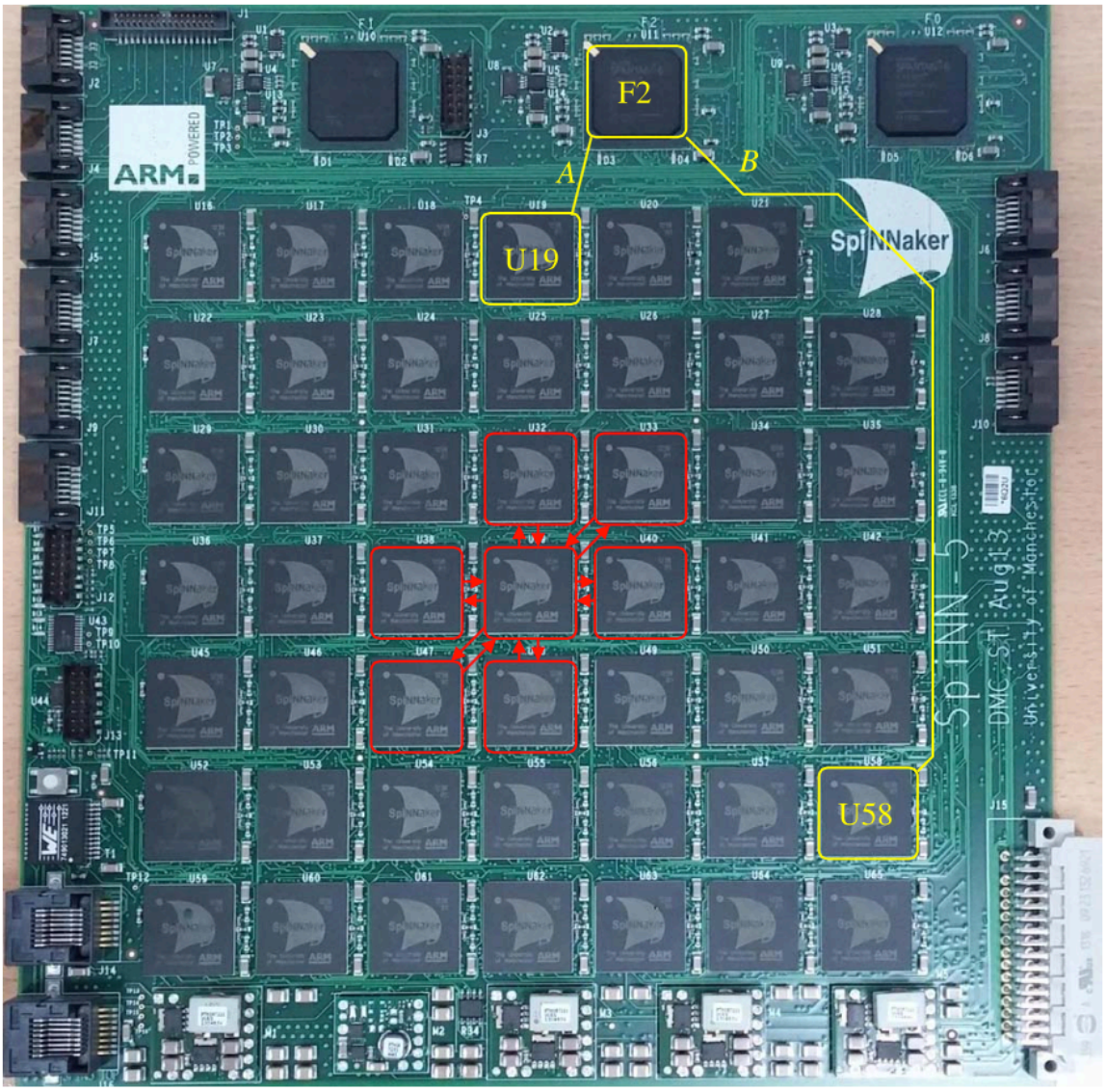


Figure 2-2: SpiNN5 board with 48 SpiNNaker chips. Lines in red illustrate inter-chip 2-of-7 links. Lines in yellow show one bidirectional 2-of-7 communication link between a Spartan6 FPGA and one of the SpiNNaker chips.

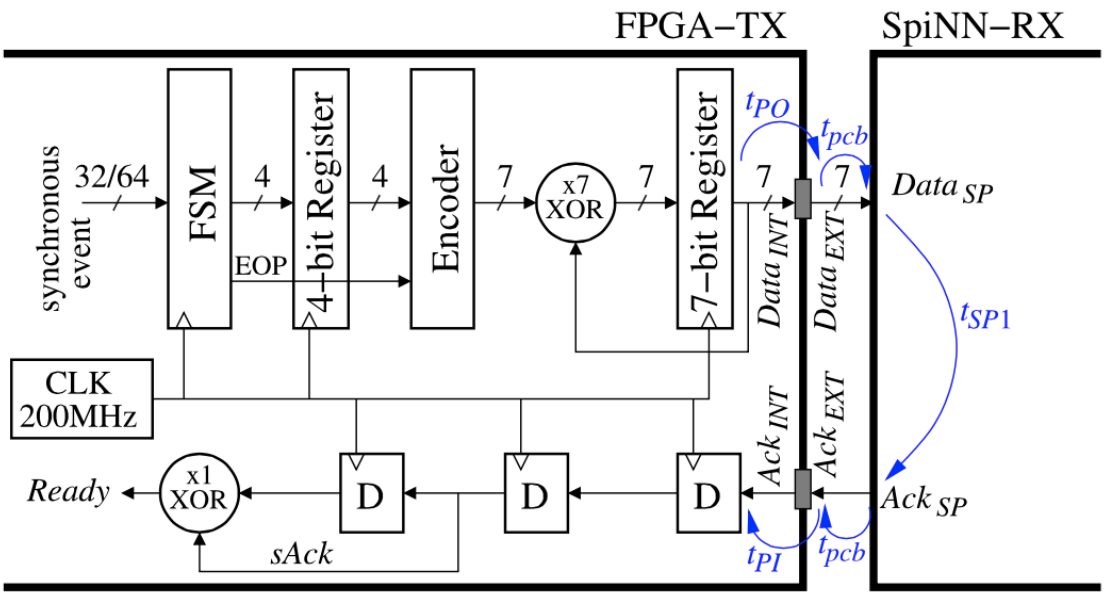


Figure 2-3: Simplified block diagram illustration of an FPGA-TX-to-SpiNNaker- RX directional link.

the speed of the reverse direction (from SpiNNaker chip to FPGA) because it would require modifying the SpiNNaker chip itself. Nonetheless, the proposed scheme can be included in future versions of the SpiNNaker chip.

2.3 Conventional Synchronization Approach

Fig. 2-3 and Fig. 2-4 show the diagram and the timing between an FPGA transmitter (TX) link side and a SpiNNaker chip receiver (RX) link side [1], using a conventional two-D-flip-flop synchronization, respectively. Short 32-bit (or long 64-bit) events are provided to a finite-state machine (FSM), which will convert them to the 11 (or 19) 4-bit symbol sequence, loading each into the 4-bit register in Fig. 2-3. After this, an "Encoder" activates the 2-of-7 bits that need to change, according to Fig. 2-1, which, after being "XOR-ed" with the previous output, provides the new output storing it in an output 7-bit register. This register holds the new 2-of-7 Data-and-Rqst $Data_{INT}$. Once it is available, it produces a delay due to output pad buffering I/O t_{PO} to go out of the FPGA as $Data_{EXT}$, plus an interchip PCB trace delay of t_{pcb} to be visible

[1]The complementary link from SpiNNaker TX to FPGA RX is not shown to save space.

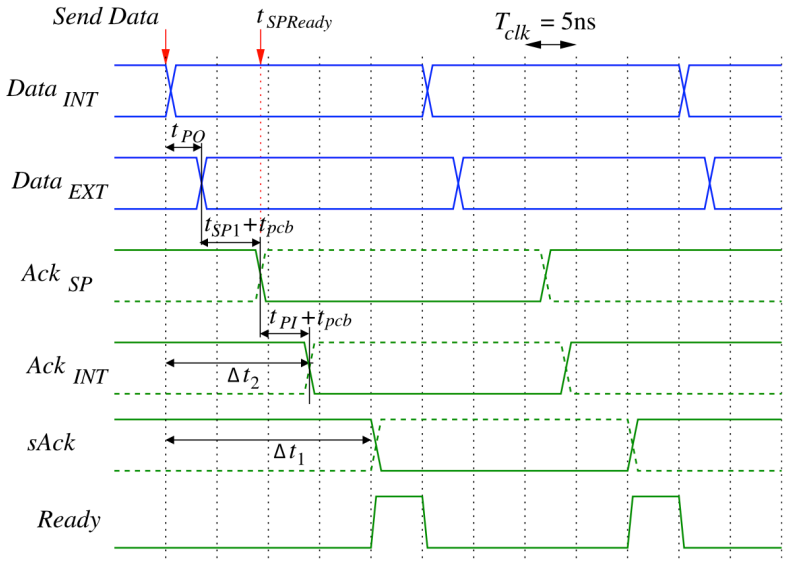


Figure 2-4: Timing waveforms illustrating NRZ handshaking and synchronization.

at the SpiNNaker chip input. The SpiNNaker chip RX port causes a delay, which is here called t_{SP1}, between detecting the 2-of-7 Data-and-Rqst until providing its acknowledge signal Ack_{SP}, which after t_pcb will make Ack_{EXT} visible at the FPGA external input. The FPGA input pad introduces an additional delay t_{PI} until the asynchronous acknowledge signal Ack_{INT} is visible internally inside the FPGA. After this, the synchronization circuit using a standard two-D-flip-flop delay line, requires two additional clock edges to make a synchronized version of the acknowledge signal $sAck$ available. At this point in time, a new $Data_{INT}$ value can be made available for the next clock edge. From the Spartan6 FPGA manufacturer specifications, we know that $t_{PI} \sim 1.2$ ns and that t_{PO} may vary between 1.7 and 5.9 ns, depending on output pad settings. For our settings, $t_{PO} \sim 3.0ns$. In Fig. 2-4, we can observe that, if $\Delta t_2 = t_{PO} + t_{SP1} + 2t_{pcb} + t_{PI}$ is between two and three clock cycles (10-15 ns), then a full symbol transaction can be done in five clock cycles (25 ns). If $10ns < \delta t_2 < 15ns$, then it implies that $5.8ns < t_{SP1} + 2t_{pcb} < 10.8ns$. Otherwise, if $10.8ns < t_{SP1} + 2t_{pcb} < 15.8ns$, then a symbol transaction would require six clock cycles (30 ns). Implementing the link in Fig. 2-4 on link A in Fig. 2-2 results in five-cycle transactions, while on link B, this results in six cycles. In summary, for the 48-chip PCB in Fig. 2-2, a symbol transaction varies between five and six clock cycles,

depending on the length of PCB traces. In the next section, we propose a method to reduce this time down to two clock cycles for both links. It requires changing the FSM in Fig. 2-3, i.e., the sender of the link. Therefore, this means that we could only test it by changing the FSM at the FPGA (the sender side). Consequently, we present results only for the case of transmitting data from the FPGA to the SpiNNaker chip.

Time t_{SP1} is typically quite stable for each SpiNNaker link, except for the cases when the interchip circuitry is sending back pressure (i.e., delaying Ack) because of internal traffic saturation.

2.4 Proposed Predictive Synchronization Scheme

The herein proposed new synchronization scheme is based on the following observation in Fig. 2-4. The SpiNNaker RX side of the link is, in principle, ready to receive a new 2-of-7 Data-and-Rqst, as soon as it has provided acknowledge signal AckSP at time $t_{SPReady}$. However, due to the synchronization with two D-flip-flops on the FPGA side, the FPGA cannot provide a new $Data_{INT}$ until four clock edges later. Here, we propose a scheme where the FPGA "learns" to forecast, reliably, the minimum number of clock cycles required to send a new symbol (without waiting to receive each symbol's synchronized acknowledge signal sACK). Nonetheless, during the multisymbol transmission of a packet, an independent process in parallel would count the total number of actual acknowledge signals received during the full packet to make sure that the full packet transaction was completed successfully.

The new proposed algorithm for the transmitter FSM in Fig. 2-3 is shown in Fig. 2-5. This FSM will send out the 11 (or 19) event/packet symbols without waiting for individual Acks from the receiver side. It will simply wait for a "Symbol Period" time (i.e., number of clock cycles) before sending the next symbol. A parallel process (not shown in the figure) will be counting the number of Ack signals received and generating an internal "Packet Ack" signal once all of them are received. The operation of the FSM in Fig. 2-5 is as follows. The first state S0 waits for a new (32 or 64-bit) event/packet. After this, the corresponding sequence of symbols must be

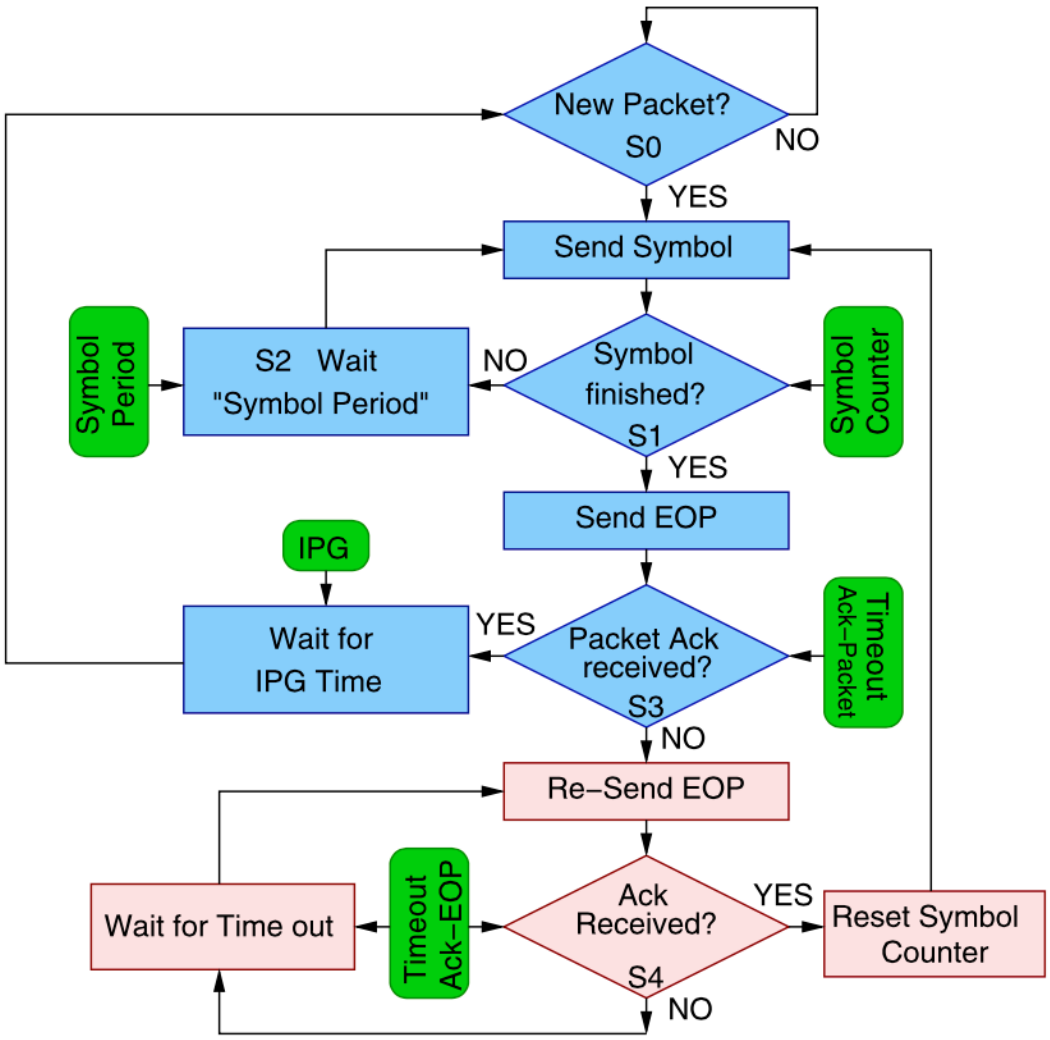


Figure 2-5: Simplified flow diagram of new transmitter FSM. Blue (medium gray) boxes correspond to failure-free packet transmission and pink (light gray) boxes to failure handling and packet retransmission, and green (dark gray) boxes indicate parameters.

sent. For this, an extra state S2 is included, which waits for a given number of clock cycles ("Symbol Period") before sending the next symbol. After sending all header and data symbols, the "End-of-Packet" (EOP) symbol is also sent. After this, the FSM enters state S3, where it waits for the internal "Packet Ack" signal. This signal is triggered only if the receiver has acknowledged all symbols sent. Once "Packet Ack" is received, an optional "Inter-Packet Gap" (IPG) wait time can be included to allow the receiver some extra time for event/packet processing. In case not all symbol Acks have been received within a given "Time-out" period, state S3 will branch out through its "Packet NOT sent" output, indicating there has been a failure in the event/packet transmission. In this case, it will try to resend the event/packet. For this, it will first send an EOP symbol to the receiver and wait for the corresponding Ack, through state S4. If this Ack is not received, then it will wait for some time to let the receiver recover and, after this, retry the transmission of an EOP symbol. This situation may occur in case the SpiNNaker internal event handling circuitry is sending back pressure (because of event traffic saturation), or there is a transient fault/disconnection in the transmission line.

At startup, there is a "learning process" in which the FSM adjusts its parameter "Symbol Period". Initially, this period is set to "1" (one clock cycle), and it will be increased progressively until reaching a stable communication. For each "Symbol Period" value, two weights are defined. The first weight is the rate of failure, and the second is the rate of success. Every time state S3 leaves through its "Yes" output, the success weight is increased. If state S3 leaves through its "NO" output, the failure counter is increased. During learning, the "Time-out" parameter values are reduced to speed up learning. The rate of convergence of this startup learning process is relatively fast, although it also depends on weight granularity and initial state. In our case, we used 8-bit weights, and both of them (success and failure) were initially set to "0". Convergence time was on the order of 500 us. After this, no more failures were detected, even when running the links for over one week. If there are transient faults during the startup learning process, it will converge to very conservative Ack/Rqst intervals. Therefore, during startup, the system and all physical connections should

be in optimum conditions.

So far, we have discussed the situation of sending events from the FPGA (as TX) to the SpiNNaker chip (as RX). In this case, it is the TX who learns to forecast the intersymbol delay, and also who detects whenever an event/packet has not been sent. To implement this forecasting/acceleration capability for the reverse direction without changing the SpiNNaker chip, we would need to change the RX side in the FPGA. For this, the receiver in the FPGA would need to send out Acks before obtaining the synchronized versions of the 2-of-7 Data-and-Rqst transitions. In case of failure, only the RX circuit in the FPGA would be aware of it, and the TX in the SpiNNaker chip would not be able to resend the event/packet. There are three obvious approaches for solving this. First, add an extra 2-of-7 command to the table in Fig. 2-1, so that the FPGA can request the retransmission of an event/packet. Second, implement this new algorithm inside the SpiNNaker chip in its TX ports. Or third, implement a slower upper layer in software to detect event/packet loss and request a new retransmission. The first two options require a redesign of the SpiNNaker chip, and we leave this as suggestions for future versions. The third solution is beyond the scope of this chapter. In the next section, we provide experimental results for the link direction from the FPGA (TX) to the SpiNNaker chip (RX).

2.5 Experimental Results

Exhaustive tests have been performed on the 48-chip SpiNNaker PCB shown in Fig. 2-2, to test the performance of packet/event communication from an FPGA to a SpiNNaker chip. The results shown here focus on two of such links: "Link-A" between FPGA "F2" and SpiNNaker chip "U19", which is one of the shortest links on the PCB, and "Link-B" between the same FPGA and chip "U58", which is one of the longest links. Experimental characterizations were performed by generating sequences of numbers with a counter on one end and checking the sequence on the other end. Failure-free transmissions were obtained after a few hundred microseconds of training, which would stay failure free for long periods (we tested for over

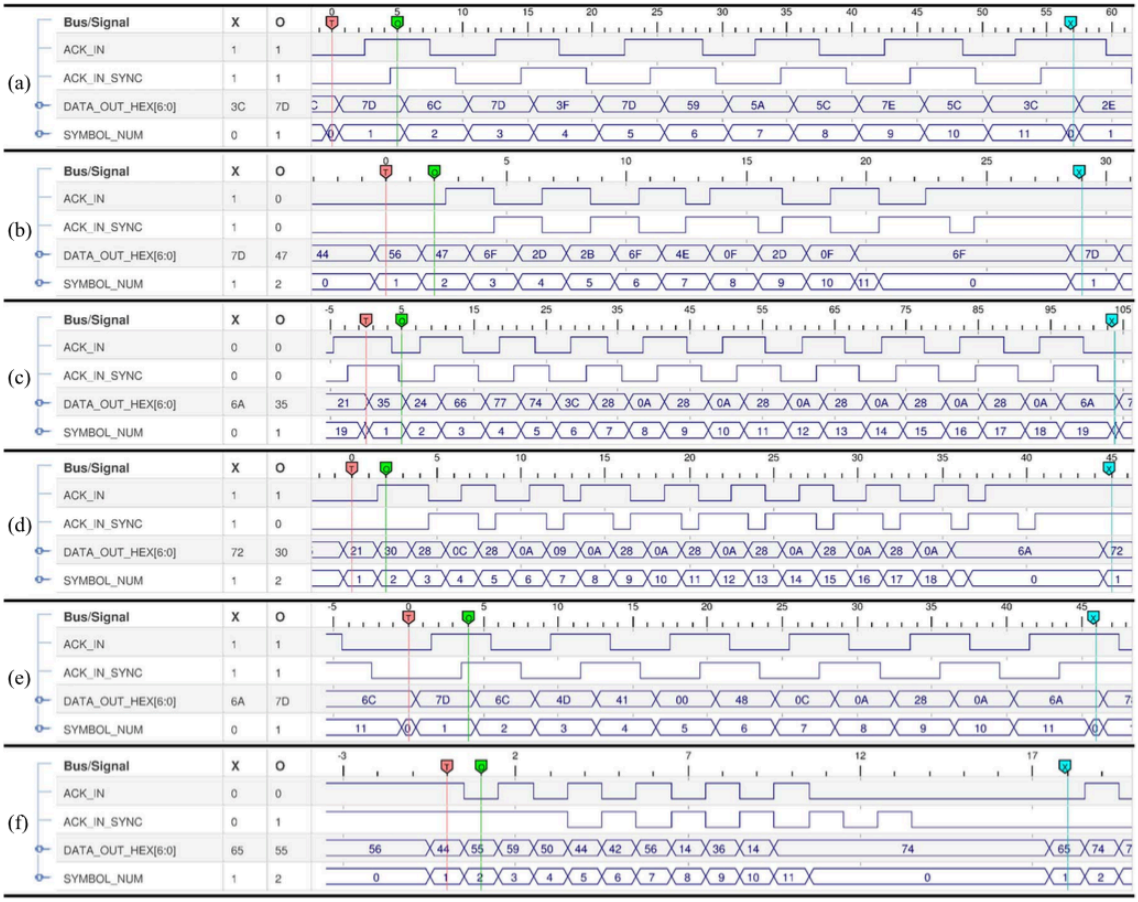


Figure 2-6: ChipScope measurements for link A with FPGA pads set to SLOW and 6mA power per pad. (a) Normal synchronization; short packet; 200-MHz clock. (b) Fast scheme; short packet; 200-MHz clock. (c) Normal synchronization; long packet; 200-MHz clock. (d) Fast scheme; long packet; 200-MHz clock. (e) Normal synchronization; short packet; 100-MHz clock. (f) Fast scheme; short packet; 100-MHz clock. Note that, in (b), ACK_IN and ACK_IN_SYNC are not exact replicas with two-clock-cycle delay. This is because ACK_IN was captured before the synchronizer and its edge must have been very close to that of the ChipScope clock.

one week). Experimental measurements were done through the use of Xilinx's built-in logic analyzer module "ChipScope". This tool allows monitoring FPGA internal signals with reference to its internal clock. For our experiments, we have set this internal clock to either 200 MHz (5-ns period) or 100 MHz (10-ns period). Fig. 2-6 shows ChipScope screen captures for different measurements. For each measurement, we show the same four signals: signal ACK_IN, which corresponds to AckINT in Fig. 2-4; ACK_IN_SYNC, which is sAck in Fig. 2-4; DATA_OUT_HEX, which is DataINT in Fig. 2-4; and SYMBOL_NUM (not shown in Fig. 2-4), which counts the symbol number within the packet/event. On the top of each subfigure, the ticks indicate clock cycle number.

Fig. 2-6(a) illustrates the case of using the conventional synchronization approach on link A for a short package with a 200 MHz clock. As can be seen, to transmit all 11 symbols, 57 clock cycles are needed, which corresponds to 3.51 Meps (mega events per second). Transmission of one symbol requires five clock cycles. Fig. 2-6(b) shows the same case, but when implementing the predictive handshaking approach. As can be seen, one 11-symbol packet needs now only 29 clock cycles, which corresponds to 6.89 Meps. Each symbol can be reliably transmitted with only two clock cycles (see signals DATA_OUT_HEX and SYMBOL_NUM), although now there is an extra seven-cycle overhead after transmitting all symbols. Interestingly, the parallel independent process in charge of counting the transitions at ACK_IN normally needs two clock cycles, although sometimes it needs one or three (see signals ACK_IN and ACK_IN_SYNC). Fig. 2-6(c) and (d) illustrates the same setup as Fig. 2-6(a) and (b) but for a long 19-symbol packet. Similarly, Fig. 2-6(e) and (f) shows the same as Fig. 2-6(a) and (b) but setting the clock to 100 MHz. This is to illustrate the situation for a slower FPGA. As can be seen, for the conventional synchronization approach, four clock cycles per symbol are required (instead of five) and 46 per packet (instead of 57). This is because the fixed delay Δt_2 in Fig. 2-4 is framed into fewer clock cycles. However, for the same reason, in the predictive handshaking approach, one symbol can be transmitted now in just one clock cycle. On the other hand, the overhead requires the same eight clock cycles; thus, the overall delay for a short-packet

		Link A								Link B							
		Short Packet				Long Packet				Short Packet				Long Packet			
		200 MHz		100MHz		200 MHz		100MHz		200 MHz		100MHz		200 MHz		100MHz	
		Normal	Fast	Normal	Fast	Normal	Fast	Normal	Fast	Normal	Fast	Normal	Fast	Normal	Fast	Normal	Fast
fast	Pck Rt	3.51	6.89	2.17	5.55	2.06	4.44	1.28	3.7	2.72	6.66	2.17	5.55	1.72	4.34	1.28	3.7
	Nc Pck	57	29	46	18	97	45	78	27	73	30	46	18	116	46	78	27
	Nc Sym	5	2	4	1	5	2	4	1	6	2	4	1	6	2	4	1
Slow	Pck Rt	3.51	6.89	2.17	5.55	1.96	4.44	1.28	3.7	2.53	6.66	2.17	5.55	1.62	4.34	1.2	3.7
	Nc Pck	57	29	46	18	102	45	78	27	79	30	46	18	125	46	83	27
	Nc Sym	5	2	4	1	5	2	4	1	6	2	4	1	6	2	4	1
quiet	Pck Rt	2.94	6.66	2.17	5.55	1.72	4.34	1.28	3.7	2.19	4.76	1.75	3.44	1.34	3.07	1.03	2.22
	Nc Pck	68	30	46	18	116	46	78	27	91	42	57	29	149	65	97	45
	Nc Sym	6	2	4	1	6	2	4	1	7	3	5	2	7	3	5	2

Figure 2-7: Measured parameters are "Pck Rt" packet rate (in mega events per second), "Nc Pck" number of clock cycles per packet, and "Nc Sym" number of clock cycles per symbol. Vertical columns show measurements for links A and B, for short packets (11 symbols) and long packets (19 symbols), at 200-MHz and 100-MHz FPGA clock frequencies, for "Normal" (conventional) synchronization scheme and for "Fast" predictive handshaking scheme. Horizontal rows are repeated for three different FPGA output pad bias settings ("current" and "slew-rate"): "fast" is 12 mA per pad with nominal 1.71-ns delay, "slow" is 6 mA per pad (recommended) with nominal 3.00-ns delay, and "quiet" is 2 mA per pad with 5.47-ns delay.

transaction is 18 cycles, resulting in a speed improvement factor of 2.55.

Fig. 2-7 shows the measured packet/event rate (Pck Rt) and the number of clock cycles per symbol (Nc Sym) and per packet/ event (Nc Pck) for all experimental setups: for links A and B, for 100-MHz and 200-MHz clock frequencies, for short and long packets, and also for three different settings of the FPGA output pads (setting SLOW with 6 mA per pad, which corresponds to all cases shown in Fig. 6; setting FAST with 12 mA per pad, and setting QUIET with 2 mA per pad). The packet transaction speed improvement varies between a factor of about 2 (1.96 for link A, short packet, 200 MHz) up to a factor of almost 3 (2.89 for link A, long packet, 100 MHz).

2.6 Conclusion

A scheme for accelerating asynchronous handshaken multisymbol packet transmissions between an asynchronous module and a synchronous one has been proposed and successfully tested on a 48-chip SpiNNaker board. The scheme exploits the fact that, within the same packet, the transaction delay per symbol remains stable and

can be "learned" by the sending circuit. Symbol Acks are counted by a separate process in parallel to verify correct packet transmission. In case of failure, the packet is resent. Exhaustive tests have been performed on the 48-chip SpiNNaker board for different PCB trace lengths, packet sizes, clock frequencies, and pad delays. Once trained, the transmission stays stable and failure free. The proposed scheme can help to improve the traffic bottleneck between SpiNNaker and PCBs, as this bandwidth is limited by the throughput between FPGA and SpiNNaker chips on board.

Chapter 3

On Multiple AER Handshaking Channels over High-Speed Bit-Serial Bi-Directional LVDS Links with Flow-Control and Clock-Correction on Commercial FPGAs for Scalable Neuromorphic Systems

This work has been published in:

A. Yousefzadeh, M. Jablonski, T. Iakymchuk, A. Linares-Barranco, A. Rosado, L. Plana, S. Temple, T. Serrano-Gotarredona, S. Furber and B. Linares-Barranco, "On Multiple AER Handshaking Channels over High-Speed Bit-Serial Bi-Directional LVDS Links with Flow-Control and Clock-Correction on Commercial FPGAs for Scalable Neuromorphic Systems" in IEEE Transactions on Biological Circuits and Systems

3.1 Abstract

Address-Event-Representation (AER) is a widely employed asynchronous technique for interchanging "neural spikes" between different hardware elements in Neuromorphic Systems. Each neuron or cell in a chip or a system is assigned an Address (or ID) which is typically communicated through a high-speed digital bus, thus time-multiplexing a high number of neural connections. Conventional AER links use parallel physical wires together with a pair of handshaking signals (Request and Acknowledge). In this chapter, we present a fully serial implementation using bidirectional SATA connectors with a pair of LVDS (low voltage differential signaling) wires for each direction. The proposed implementation can multiplex a number of conventional parallel AER links for each physical LVDS connection. It uses flow control, clock correction, and byte alignment techniques to transmit 32-bit address events reliably over multiplexed serial connections. The setup has been tested using commercial Spartan6 FPGAs attaining a maximum event transmission speed of 75Meps (Mega events per second) for 32-bit events at a line rate of 3.0Gbps. Full HDL codes (VHDL/Verilog) and example demonstration codes for the SpiNNaker platform are freely available for the academic purpose.

3.2 Introduction

Address Event Representation (AER) is now a fairly popular "virtual wiring" technique adopted by many neuromorphic hardware engineers to interconnect spiking neuromorphic systems [16, 24, 13, 25, 26, 27, 28, 29, 30, 3, 31, 32]. Digital inter-chip communication is several orders of magnitude faster than firing frequencies of biological neurons. This is exploited in AER to time-multiplex numerous synaptic connections between neurons on a high-speed digital bus. In AER, whenever a spiking neuron in a chip (or module) generates a spike, its "address" (or any given ID) is written on a high-speed digital bus and sent to the receiving neurons in one or more destination modules/chips. AER started out as a point-to-point protocol for inter-

connecting neurons from one chip with those on another chip using a hand-shaken parallel digital bus with a fixed number of bits [16, 24, 13]. As neuromorphic systems have been scaling up in size and complexity over the years, researchers have developed more complex and smarter AER "variations" to improve efficiency, reconfigurability and reliability. It became apparent very early on that bulkiness of the original parallel-AER (pAER) bus would limit the scalability of AER systems to arbitrary sizes, and researchers started looking at serial connectivity options [33]. In 2004 Boahen proposed a word-serial AER link to reduce the number of parallel wires [34]. The 48-chip SpiNNaker PCB [35] uses six bidirectional 8-wire 4-bit word serial AER-type links per chip, employing a highly power-efficient delay-insensitive 2-of-7 NRZ protocol [10] to asynchronously interchange 32-bit events between the chips, each chip holding 18 ARM968 integer-arithmetic cores.

Fully bit-serial Low-Voltage-Differential-Signaling (LVDS) [36] AER links have also attracted the attention of some neuromorphic engineers, as they allow for multi-gigabit-per-second communication speeds using only one pair of wires. Since just two unidirectional differential wires are available, it is not straightforward how to implement a handshaking protocol per event transmission or a flow-control scheme to signal data congestion on the receiver side. Miro et al. [37] and Berge et al. [38] experimented with fully bit-serial links. The former with off-the-shelf MAXIM serdes components, showing measurements that revealed the links could operate up to 40Meps (Mega-events-per-second) for 16-bit AEs (Address Events). The latter with the 2.5Gbps (Giga-bit-per-second) LVDS Rocket-IO IP blocks available in the VirtexII-Pro Xilinx FPGAs, achieving 41.66Meps for 16-bit AEs, with 8b/10b encoding for byte-alignment comma transmission when the channel is idle. However, in both cases, the FSMs (Finite State Machines) implemented did not include any hand-shaking or flow-control mechanisms to avoid data loss if the event consumer system on the receiver side was temporarily slower than the event generator system on the transmitter side. Fasnacht et al. [39] reported a bit-serial interface based on off-the-shelf commercial 16-bit Serializer/Deserializer components (TLK 2501/3101) connected to a Spartan3E parallel event AER processor and using a second, reverse,

LVDS link for flow control signaling. The bit-serial link could operate at line speeds of 2.5Gbps (TLK 2501) or 3.125Gbps (TLK 3101) and also used 8b/10b encoding to allow for idle commas. On the reverse LVDS link, the receiver put a square wave whose frequency signaled whether to stop or resume event transmission from the sender. Using this setup, the link could transmit 32-bit events at a maximum rate of 62.5Meps (for 2.5Gbps) or 78.125Meps (for 3.125Gbps), at the cost of sacrificing one reverse LVDS link for flow control. Zamarreno et al. [40] developed a bi-directional LVDS link using Virtex6 FPGA Rocket-I/O IPs. In this scheme, two 2.5Gbps LVDS links were used to communicate 32-bit events in each direction with 8b/10b encoding. Flow-control in each direction was implemented by inserting special control symbols in the opposite direction link to toggle between the stop and resume states. However, this design did not include any clock-correction support, thus hampering scalability by limiting this approach to situations in which all FPGAs use the same physical reference clock.

Fully ASIC designs for 32-bit event transmission over 1GHz bandwidth LVDS links have also been reported recently using either current-mode [41] or voltage-mode [42] drivers [1]. These use four wires: two for high-speed 1GHz bit-serial LVDS data transmission and two for hand-shaking. The extra hand-shake wires allow the drivers to be turned ON/OFF during inter-event pauses (with nano-second latencies), resulting in power consumption proportional to the information transmission rate. This way, no idle commas have to be transmitted, and 8b/10b encoding is not necessary. On the down-side, Manchester encoding is required to allow for simultaneous data and clock transmission per symbol [43], reducing effective data transmission to half of maximum physical rate. Nevertheless, maximum 32-bit event rates of up to 15Meps have been measured over a channel with a physical bandwidth 1.4GHz.

All previous asynchronous pAER to bit-serial AER conversion schemes operate correctly if there is only one clock domain at each synchronous side of each link. This is usually the case in fully ASIC designs, where all AER processing is fully

[1] Although the CMOS process used was quite old with a large minimum feature size (0.35um), the drivers achieved an impressive performance of up to 1.4GHz bandwidth.

asynchronous, and there is only one local clock per LVDS Transmitter/Receiver link located at the transmitter side (the receiver uses the clock extracted from the transmission). In the case of the reported FPGA interfaces with unidirectional links, the situation is similar, because usually the FPGA synchronous circuit only implements one state machine to interface with an asynchronous hand-shaken pAER port. One single, isolated clock domain can therefore be used per link. In the case of bi-directional links, the situation is not so straightforward when using commercial FPGAs with bit-serial IPs, and two interfering clock domains might be required. The problem becomes even worse when interconnecting multiple FPGAs, each with its own synchronous event processing subsystem and multiple bidirectional LVDS links per FPGA. In this case, each FPGA will have one or more local clock domains, which may interfere with the clock domains of neighboring FPGAs. Under these circumstances, it is necessary to use some clock correction technique to compensate for clock frequency/phase drifts and avoid sudden byte misalignment problems and data loss. In a preliminary work [6], we used the elastic buffers available in Xilinx Rocket-I/O serial LVDS IPs, although the test involved a fully synchronous handshake-less system deployed over two FPGAs. Each link was unidirectional since the reverse LVDS path was fully occupied handling flow-control. More recently, the SpiNNaker team has developed bidirectional bit-serial LVDS links [44] to bundle eight 2-of-7 AER multi-symbol inter-SpiNNaker-chip links [10] into one bit-serial SATA link. This scheme, designed to interconnect multiple (up to 1200) 48-chip SpiNNaker Boards [1], uses flow-control, clock correction, and a sophisticated framing protocol that samples the eight channels and performs CRCs (Cyclic Redundancy Checks) to improve reliability. However, this introduces extra overheads, theoretically limiting the maximum throughput for each of the eight channels to $50/8 = 6.125$Meps.

In this chapter, we present an extended version of the solution reported earlier [6], with a fully bidirectional bit-serial LVDS communication capability, a token-based flow-control protocol, a clock correction capability, and a robust interface with conventional parallel AER ports (like those used in AER sensor chips) using 4-phase asynchronous handshaking. At an LVDS line transmission rate of 3.0Gbps, it is

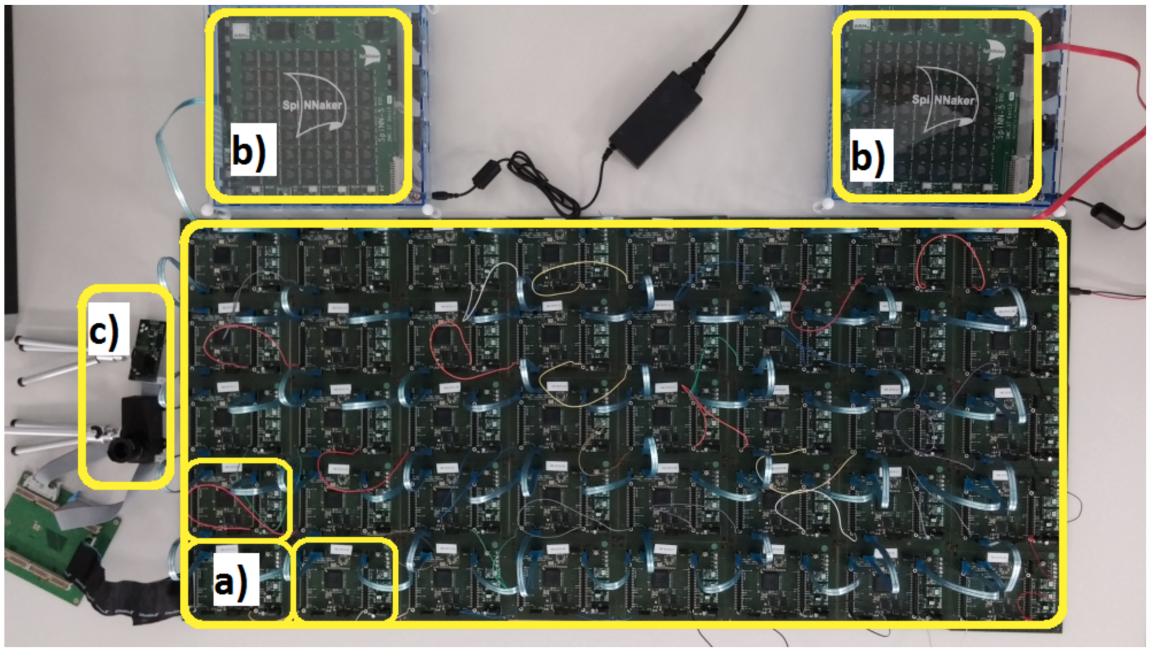


Figure 3-1: Example heterogeneous neuromorphic sensing and computing system consisting of a) 5x8 grid of FPGA-based AER-Node boards connected to each other via four bi-directional LVDS links, b) two SpiNNaker boards comprising each 48 SpiNNaker chips (each with 18 ARM CPUs) organized into grids connected via 2-of-7 asynchronous parallel buses with 2-phase handshaking, and c) two asynchronous artificial retina vision sensors connected via parallel AER 16-bit external buses with 4-phase handshaking.

possible to achieve 32-bit transmission at a sustained 75.0Meps, as shown later in the Experimental Results section. Fig. 3-1 shows an example target setup consisting of an array of 40 AER-Node Boards [6] all interconnected via SATA wires to their neighbors, to two 48-chip SpiNNaker Boards [1] and to two AER retina sensors [2].

The presented serial intercommunication protocol extensively exploits the Spartan6 GTP transceiver instance using specific configurations together with user-designed TX and RX blocks for low bandwidth overhead intercommunication of heterogeneous components (multiple clock-domain synchronous modules with fully asynchronous sensors), while multiplexing multiple AER channels with independent flow control with minimum latency and almost maximum physical channel throughput. This results in very efficient assembly capability of heterogeneous neuromorphic systems. Application examples are illustrated at the end of the chapter.

The chapter is structured as follows. Section 3.3 explains how we used the 8b/10b encoding scheme for bi-directional token-based flow-control and 32-bit event alignment. Section 3.4 examines in detail the problem of multiple interfering clock domains when using multiple FPGAs each with multiple bidirectional links, and how to overcome it using elastic buffers and clock-correction. Finally, Section 3.5 provides experimental results.

3.3 Bi-Directional Token-Based Flow-Control and Event-Alignment Using 8B/10B Encoding

When transmitting GHz range bit-serial data over a single lane (either differentially over a pair of wires as in LVDS, or using one single wire), the transmitted stream must include not only the serial data itself but also sufficient clues to allow clock recovery at the receiver [2]. The Manchester encoding scheme [45] encodes bits '0' and '1' as two different transitions: from high-to-low or low-to-high. This way, for each

[2]In the case of multiple bit-serial lanes operating in parallel (which is not the case considered in this chapter), one lane can be dedicated to transmitting the reference clock. In this case, the lanes have to be very well matched to avoid excessive skew.

Table 3.1: Control Commas in 8b/10b Coding

	8-bit in			10-bit out	
Name	DEC	HEX	BIN	RD = -1	RD = +1
K28.0	28	1C	000 11100	001111 0100	110000 1011
K28.1	60	3C	001 11100	001111 1001	110000 0110
K28.2	92	5C	010 11100	001111 0101	110000 1010
K28.3	124	7C	011 11100	001111 0011	110000 1100
K28.4	156	9C	100 11100	001111 0010	110000 1101
K28.5	188	BC	101 11100	001111 1010	110000 0101
K28.6	220	DC	110 11100	001111 0110	110000 1001
K28.7	252	FC	111 11100	001111 1000	110000 0111
K23.7	247	F7	111 10111	111010 1000	000101 0111
K27.7	251	FB	111 11011	110110 1000	001001 0111
K29.7	253	FD	111 11101	101110 1000	010001 0111
K30.7	254	FE	111 11110	011110 1000	100001 0111

symbol (either ‘0’ or ‘1’) there is always a physical transition that makes it possible to recover the clock instantly during symbol extraction. The drawback is that the data rate is only half the channel’s maximum possible physical rate.

8b/10b encoding [46] overcomes this severe limitation by mapping 8-bit words to 10-bit symbols, assuring enough state changes for reasonable clock recovery, while achieving DC balance. The difference between the counts of ‘1s’ and ‘0s’ (called "disparity") in a string of at least 20 bits is no more than two, and there are not more than five ‘1s’ or ‘0s’ in a row[3]. Consequently, this scheme allows effective data transmission at a rate of 80% of the physical channel bandwidth. Clock recovery from the bit stream is not instantaneous and is normally performed by complex PLL (phase locked loop) circuits at the receiver, which may require sequences in the order of thousands of bits to lock to the clock frequency. Also, to keep the PLLs locked all the time, the channel needs to keep transmitting symbols even when no information has to be sent. Fortunately, when mapping the 256 8-bit symbols into 10-bit symbols,

[3]Since shuffling five ‘0s’, and five ‘1s’ (perfectly DC balanced 10-bit symbols) yields less than 256 10-bit symbols, some 8-bit symbols are mapped into two unbalanced 10-bit symbols (one with four ‘0s’ and six ‘1s’ and the other with six ‘0s’ and four ‘1s’). A counter keeps track of the "running disparity" (RD) between ‘1s’ and ‘0s’ and picks one of the two possible 10-bit symbols so that the RD is compensated.

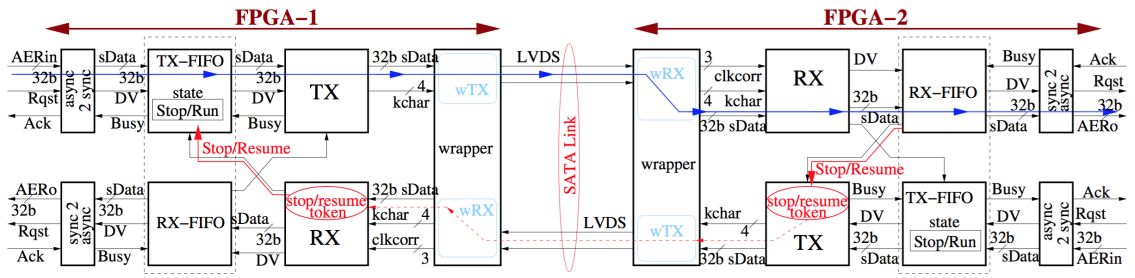


Figure 3-2: 2-FPGA LVDS Bi-Directional AER Communication Link with Flow-Control to/from four pAER ports. The figure illustrates how stop/resume control tokens are exchanged via the complementary channel to achieve flow control.

there are more than 256 10-bit symbols that satisfy the DC balance and state change restrictions. These extra 10-bit symbols (which do not have a corresponding 8-bit symbol) can be used as "control symbols", also called "control commas". One of them can be used as the "idle comma" to keep transmitting data over the channel and keep the PLLs locked. Table 3.1 shows the 12 possible control symbols allowed in 8b/10b coding. The first column shows the names given to these control symbols. The next three columns show their 8-bit representation format (in DEC, HEX, and BIN), and the last two columns show their 10-bit binary representation depending on the running disparity RD4. If K28.7 (FC) is not used, it is easy to use K28.1 (3C) and K28.5 (BC) for synchronizing byte alignment within the bit-stream: that is to say, to find the start/end of the consecutive 10- bit symbols. This is because the unique sequences '0011111' or '1100000' cannot be found at any bit position within any combination of normal codes.

Fig. 3-2 shows a block diagram of a bi-directional AER link using two FPGA PCBs connected by one single SATA cable (containing two differential pairs of signal wires and three ground wires). Each FPGA connects to one hand-shaken 32-bit pAER sender and one pAER receiver. The "wrapper" block is an IP block generated by Xilinx Core Generator. It contains a wrapper transmitter sub-block wTX and a wrapper receiver sub-block wRX. wTX takes a 32-bit clock-synchronous "sDATA" word and a 4-bit "k-char" word as input. Each of the four "k-char" bits indicate whether the corresponding four bytes in the 32-bit "sDATA" word are control comma

bytes or regular user data bytes. In our case, the selected LVDS line rates were 3.0Gbps, obtained from a low jitter differential 150MHz reference Xtal oscillator on the FPGA PCB. This means that the 32-bit data, transformed into a 40-bit sequence by the 8b/10b code, needed $40/(3109Hz) = 13.3ns$ to be transmitted. The wrapper provides two reference clocks for the user circuitry: one at frequency f = 1/13.3ns = 75.0MHz, at the rising edges of which the user circuitry has to provide the 32-bit parallel "sDATA" word and the corresponding 4-bit "k-char" word; and the other at frequency 4f = 300MHz (because the 32-bit "sDATA" word contains 4 bytes) and synchronized to clock f. The user designed circuitry within each FPGA in Fig. 3-2 has 6 blocks:

1. **Asynchronous to Synchronous Converter Block (async2sync).** This block handles asynchronous hand-shaking with the 32-bit pAER input port and synchronization between the asynchronous and synchronous domains (or two synchronous domains driven by unrelated clocks). It also provides a 32-bit parallel synchronous version "sDATA" word clock-synchronized with its corresponding "Data Valid" (DV) control signal.

2. **TransmitterFIFOBlock(TX-FIFO).** This block holds a "Stop/Run" state for the transmission of data which is used by the flow control protocol discussed later. It also includes a small FIFO for transient data storage. The need for this FIFO will become more apparent later when we address extension to multi-channel multiplexing.

3. **Transmitter Block (TX).** For each clock cycle this provides the 32-bit "sDATA" word and the 4-bit "k- char" word required by the "wrapper" (wTX), generates start-up byte alignment sequences, and inserts control symbols for flow control, clock correction, idle commas, or periodic commas for alignment.

4. **Receiver Block (RX).** This block receives and separates data and control commas. Data symbols are sent to the RX-FIFO block, while control commas are interpreted and executed for proper flow control, word (re)alignment, and clock correction.

5. **Receiver FIFO block (RX-FIFO).** This block accumulates 32-bit synchronous data symbols from the RX block into a FIFO register, while it empties the FIFO by sending data out to the "sync2async" block. If the FIFO fills up above a certain threshold, it will trigger the flow control mechanism, as explained below.

6. **Synchronous to Asynchronous Block (sync2async).** This block handles asynchronous handshaking with the 32-bit pAER output port and synchronization with the synchronous data "sDATA" clock domain.

3.3.1 Four-Byte Alignment

At start-up, and after all PLLs are properly locked, each TX block will send a sequence of 1024 32-bit comma words (3C BC BC BC) for event word alignment at the RX block on the other side of the link. The Xilinx wrapper IP includes internal circuitry for aligning bytes, using either control comma K28.5 (BC) or K28.1 (3C). In our implementation, we selected K28.5 (BC) as the byte alignment comma to be recognized by receiver wRX in the wrapper. As soon as the wrapper receiver circuit detects this comma, from then on all bit sequences will be aligned to recognizable bytes. In our case, our event or data words consisted of 4 bytes. We therefore needed to add extra circuitry in the user receiver to properly align 4-byte events. In principle, the transmitted control word for alignment "3C BC BC BC" can be received on the receiver side (after correct byte alignment) with four possible offsets: "3C BC BC BC" (offset = 0), "BC 3C BC BC" (offset = 1), "BC BC 3C BC" (offset = 2), or "BC BC BC 3C" (offset = 3). Depending on which of these four possibilities is obtained, the receiver will set its initial "offset" to either '0', '1', '2', or '3', respectively. Once the offset is known, the four-byte sequence is then correctly reconstructed by holding the 4-byte data in two consecutive registers, as shown in Fig. 3-3. First, the 32-bit data read from the wrapper is stored in register 'datain_1', and in the next clock cycle, it is copied from register 'datain_1' to register 'datain_2'. Depending on the running value of "offset", four bytes from 'datain_1' and 'datain_2' are copied to output register 'out_data', as shown in Fig. 3-3, to reconstruct the correct 4-byte

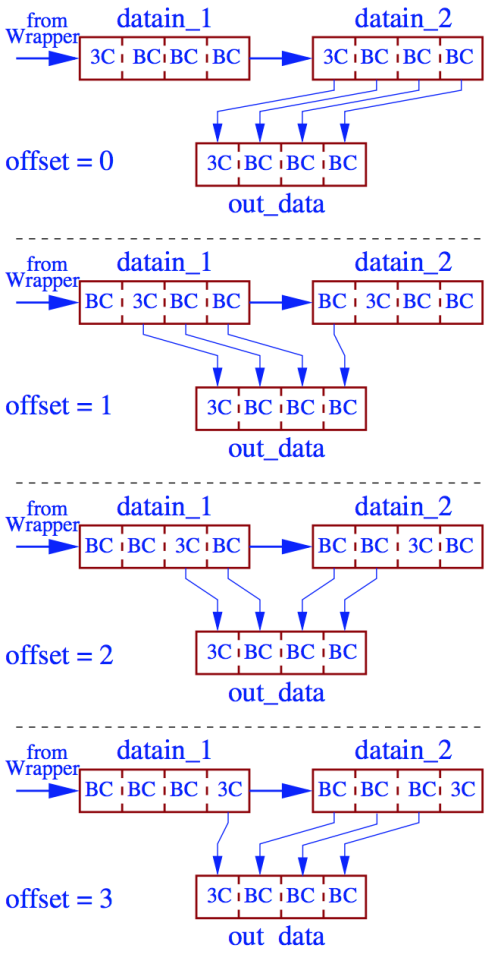


Figure 3-3: Illustration of 4-byte re-alignment depending on running offset. The four different offset cases are shown and how the output data is assembled from the flowing input data.

event sequence. Once the running offset is detected, it will remain fixed for the forthcoming data stream sequence, unless there is a clock correction (this is discussed in Section 3.4). In this case, the running offset will be updated. For now, let us assume that, after the initial 4-byte alignment, all 32-bit data and control commas sent from the sender side remain aligned and are correctly received on the receiver side. Implementation of the flow control mechanism is explained below.

3.3.2 Flow Control

With reference to Fig. 3-2, let us assume that we are sending events from the left-side pAER sender (top left of FPGA-1 in Fig. 3-2) to the right-side pAER receiver (top right of FPGA-2 in Fig. 3-2). Event communication and flow control in the opposite direction is fully symmetric and simultaneous. If no control token needs to be communicated, the default operation is as follows: the async2sync block (see details in Subsection II-C) acknowledges incoming 32-bit AER events, synchronizes them to the local f = 75MHz clock and transfers the synchronized event "sDATA" in a single clock cycle to the "TX-FIFO" block. By default, this block is in the "Run" state, in which the 32-bit "sDATA" event is transferred to the top left "TX" block in one clock cycle. This TX block will send the 32-bit "sDATA" event to the input port of the "wrapper" (wTX) while setting k-char=‘0000’ (that is to say, none of the four bytes is a control comma). This way, the events read from the pAER sender will be streamed one after another over the bit-serial LVDS line to the destination FPGA. As will be explained below, the async2sync block needs several clock cycles to acquire one 32-bit event, while the other synchronous blocks need only one clock cycle to transfer "sDATA". Since the "wrapper" needs to read 32-bit data at every f = 75MHz rising clock edge, the TX block will insert "idle commas" (3C BC BC BC) whenever there is no actual event to be transmitted. On the receiver FPGA, the "wrapper" (wRX) will provide reconstructed 32-bit "sDATA" words together with their corresponding 4-bit "k-char" words. Depending on the running "offset" value, the RX block will correctly align the four bytes for each event, and send the correctly assembled 4-byte 32-bit events to the RX-FIFO block. After the sync2async block, the top right pAER

receiver will read all these events from RX-FIFO. If the top right pAER receiver temporarily reads events slower than the top left pAER sender, the RX-FIFO block will accumulate an increasing number of events and eventually fill up. To overcome this, a flow control mechanism is required. We defined two threshold levels for the RX-FIFO block: a "stop threshold" and a "resume threshold". If the FIFO is filled above the "stop threshold", it asserts a "STOP" bit signal for the bottom right TX block (in FPGA-2) of the reverse transmission link. As soon as the TX block detects this active "STOP" signal, it will send a 32-bit "stop token" control word (01 1C 1C 1C), together with k-char=‘0111’, through the reverse LVDS channel to the bottom left RX block in FPGA- 1. This RX block will then activate a "STOP" signal for the top left TX-FIFO block in FPGA-1, which will then change its state to "Stop" and will refuse to accept new events (by activating "Busy"). Consequently, from now on no more events will be transmitted on the upper forward channel from left to right, and the RX-FIFO in FPGA-2 will be gradually emptied. When the RX-FIFO content crosses the "resume threshold", it will de-assert the "STOP" signal and the TX block in FPGA-2 will send a 32-bit "resume token" control word (00 1C 1C 1C), together with k-char=‘0111’, through the reverse channel to the RX block in FPGA-1. This block will de-assert the "Stop/Resume" bit signal and the TX-FIFO block in FPGA-1 will resume accepting events. Token control word transmission has higher priority than normal event word transmission, so the TX blocks will momentarily stop accepting new events (by activating the "Busy" signal) until the 32-bit token is sent. This will introduce a latency of just one clock cycle in regular event data transmission.

The stop/resume thresholds can be adjusted experimentally for worst case delay. A simple method to measure this delay is by using two GTP transceivers in one FPGA and a counter to obtain the number of clock cycles between sending and receiving a given event. We also observed in previous research [47] that slight differences ($\sim 30ppm$) in Xtal oscillator frequencies can also impact system behavior by introducing asymmetries in the data rates of the links, therefore justifying also the need for adjusting these thresholds experimentally.

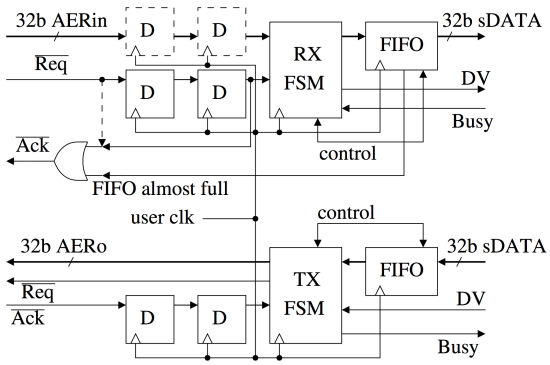


Figure 3-4: Simplified diagram of async2sync (top) and sync2async (bottom) blocks. The figure comprises the standard scheme (solid lines) together with a proposed accelerated scheme (dashed lines).

3.3.3 Asynchronous to synchronous domain interfacing

Fig. 3-4 shows simplified diagrams for the Asynchronous to Synchronous (top) and Synchronous to Asynchronous (bottom) blocks in Fig. 3-2. These blocks interface between the synchronous circuits in one FPGA clock domain and external AER circuits, which may be fully asynchronous or driven by other domain clocks. In either case, synchronization is required. Here we used the conventional two D-flip-flop scheme to read in and synchronize the external incoming handshake signals (either Req for the top part, or Ack for the bottom part). We have performed tests using two schemes: (a) The conventional two D-Flip-Flop scheme [48] shown in Fig. 3-4 when ignoring the wires and blocks with dashed lines, and (b) an accelerated scheme that includes the wires and blocks with dashed lines. The conventional scheme requires on average 12 clock cycles for one event transaction, while the accelerated one needs on average six clock cycles.

1. *Conventional Scheme:* The conventional synchronization scheme introduces a delay of an extra two clock cycles per handshake signal (Req on the receiver and Ack on the sender). The FIFOs are both small (4 registers). On the synchronous side (right) of the local clock domain, data is transferred with a single clock cycle transaction whenever signal DV (data valid) is activated while signal Busy is inactive. On the left side, data is transferred using 4-phase handshaking. Under

these circumstances, one event transaction requires 10 to 12 clock cycles [48].

2. *Accelerated Scheme:* For the accelerated scheme we have to impose some restrictions on the sender, the delays of the interconnection lines, and the relative difference between the clock frequencies of sender and receiver circuits. For the sender, the Req signal has to be set one (sender) clock cycle after the data lines, the jitter of the interconnection lines has to be negligible with respect to clock cycles, and the difference in clock frequencies between sender and receiver cannot be greater than a factor two. Under these circumstances, one event transaction can be performed in either 5 or 6 transmitter clock cycles. The scheme works as follows. For the async2sync receiver interface (top in Fig. 3-4), the (active low) Ack signal is formed by the OR of signal (active high) "FIFO almost full" and (active low) Req. Consequently, the returning Ack signal needs to be synchronized on the other side, and this is done using another two D-flip-flop scheme. The receiver also synchronizes all data lines, in order to delay them two clock cycles. It is well known that, in general, synchronizing parallel data lines does not work [49]. This is because each data bit can be captured at different receiver clock edges due to jitter in the sender data edges, jitter in the receiver data latches clock edges, different delays of the data bit lines or different threshold levels of the data bit latches. However, if it is guaranteed that at the receiver side (a) Req always arrives after all data lines have stabilized, and (b) Req and all data lines stay stable for at least two clock cycles, then the synchronized version of Req will capture all data bits correctly. The chance of metastability, however, is multiplied by the number data bits plus one (33 in our case). Nonetheless, for the Spartan6 specifications, this chance is still several years. In the experimental results (Section 3.6) we show in hardware verifications of both schemes, the conventional one and the accelerated one, tested for over 65 hours without any transmission errors.

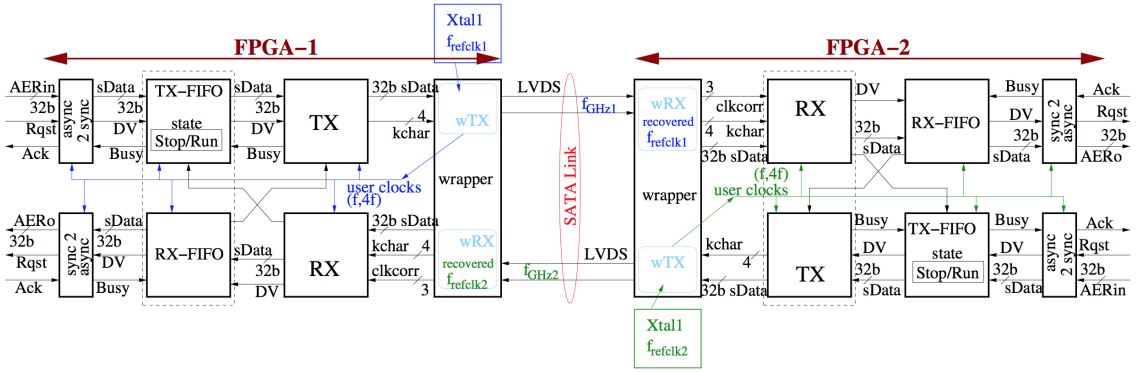


Figure 3-5: 2-FPGA LVDS Bi-Directional AER Communication Link highlighting the Reference Xtal Oscillator in each FPGA. Xtal frequencies may differ by a few ppms, thus requiring clock correction techniques for reliable communications.

3.4 Clock Correction

3.4.1 The Problem of Multiple Clock Domains

Fig. 3-5 shows the bi-directional LVDS link between the two FPGA PCBs in Fig. 3-2, this time highlighting the reference Xtal oscillator in each FPGA PCB. Each PCB has its own Xtal oscillator which generates a low-jitter reference clock (in our case, 150MHz). In FPGA-1 PCB, reference clock $f_{refclk1}$ will be up-converted by special circuitry inside the wrapper to the line frequency f_{GHz1} used by the LVDS transmission (wTX) lane (typically in the range of a few GHz). Extra clock management circuitry also provides two additional reference clocks of frequencies f and 4f (when 32-bit 4-byte data is read per clock cycle)[4]. Wrapper input data (32-bit "DATA" and 4-bit "kchar") is read at the rising edges of f. Clock f_{GHz1} is used to encode the bit-serial data stream through the top LVDS lane in Fig. 3-5. The receiver circuitry inside the wrapper (wRX) in FPGA-2 PCB will recover the f_{GHz1} clock to decode the serial data and convert it back to parallel 32-bit "DATA" words. Let us call this recovered clock f_{rGHz1}. It will have exactly the same frequency as f_{GHz1} but with totally uncorrelated phases and higher jitter. On the receiver side it is possible to down-convert the recovered clock f_{rGHz1} to the original frequency of $f_{refclk1}$. Clock

[4]Frequencies f_{GHz} and f are related by $f_{GHz} = 10 \times nbpe \times f$, where n_{bpe} is the number of bytes per event. Throughout this paper $n_{bpe} = 4$, $f_{GHz} = 3GHz$, $4f = 300MHz$, and $f = 75MHz$.

signals $f_{refclk1}$, f, $4f$, f_{GHz1} in FPGA-1 PCB, and recovered clocks f_{rGHz1} and $f_{refclk1}$ in FPGA-2 PCB are all mutually synchronous because they are all from the same reference Xtal1 oscillator.

Likewise, similar clocks are generated derived from the Xtal2 oscillator in FPGA-2 PCB. Both Xtal oscillators (Xtal1 and Xtal2) must have the same frequency. However, there is always a small frequency difference (typically approximately +/-100ppm) between reference clock sources. As a result, each wrapper uses a slightly different frequency for its transmit data path (wTX) and its receive data path (wRX). There are therefore two independent clock domains, and at some interface, they are bound to interfere with each other. This situation can be dealt with using a clock correction technique.

Alternatively, instead of clock correction, an attempt can be made to extend the same clock domain to all circuitries by propagating the clock recovered at wRX. The drawback of this approach is that the clock recovered by a receiver in a wrapper has higher jitter than the original clock, resulting in less reliable communication. If, instead of having just one bi-directional link (as in Fig. 3-5), there are more links per FPGA-PCB and many PCBs are interconnected, then it will be impractical to propagate a single Xtal reference clock to all PCBs through a sequence of clock up-conversions and recovery down-conversions. This would progressively degrade clock jitter and ultimately render the links unusable. In this case, clock correction techniques offer a robust, reliable solution. The use of one such method, available in some FPGAs, is detailed below.

3.4.2 Clock Correction Implementation

Consider the situation in Fig. 3-6(a), which shows a FPGA with four bi-directional LVDS links (8 LVDS lanes). This FPGA can be thought of as being held in one FPGA-PCB with its reference Xtal oscillator. Many of these single-FPGA PCBs could be interconnected in a mesh-like fashion using bi-directional LVDS links to form a 2D array of PCBs, as shown in Fig. 3-1. Each FPGA contains 4 "wrapper" circuits (like the ones discussed in Section 3.3 and shown in Fig. 3-2 and Fig. 3-5),

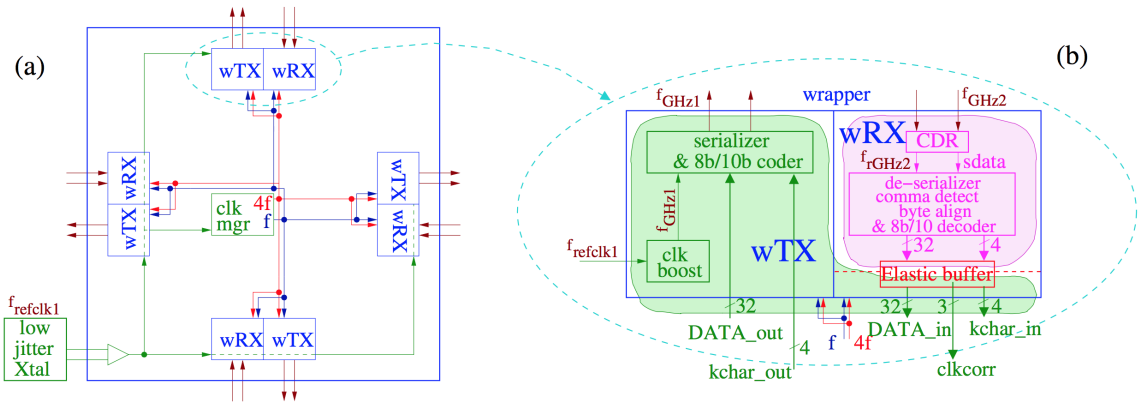


Figure 3-6: (a) FPGA PCB with four bi-drectional LVDS bit-serial links. (b) Detail of wrapper transmitter (wTX) and receiver (wRX) sub-blocks.

one per bi-directional LVDS link.

Each FPGA-PCB has one low jitter Xtal oscillator (150MHz in our case). These low jitter oscillators usually provide a differential output signal, which is converted to a single-ended signal inside the FPGA with specially dedicated low-jitter buffers. This reference clock is then routed to all wrapper transmitter (wTX) subcircuits, where it is up-converted to a clock signal at line frequency f_{GHzi} (in the range of 1-3.2GHz for Spartan-6). The reference clock $f_{refclki}$ is also routed to a "clock manager module" (clk mgr in Fig. 3-6(a)), where two synchronized clocks of frequency f and $4f$ are provided. These two synchronized clocks are routed to the wrappers and are also available for user logic.

Fig. 3-6(b) shows the internal structure of the wrapper in more detail (although still overly simplified). In the wrapper transmitter sub-block (wTX), "DATA_out" (32-bit events or control commas) is read synchronously at the rising edges of clock f, together with its corresponding 4-bit "kchar_out" word. This parallel 32-bit word is then converted byte by byte, using 8b/10b encoding, to give a 40-bit parallel word, which is then serialized into a bit-stream using the up-converted clock frequency f_{GHz1}, and sent through the appropriate LVDS driver circuits to a differential pair of wires. Fig. 3-6(b) also shows the simplified internal structure of the "wrapper receiver sub-block" (wRX). A CDR (Clock and Data Recovery) circuit receives the LVDS bit-serial stream of input data, extracting a recovered clock f_{rGHz2}. A deseri-

alizer circuit then converts this bit-stream into a sequence of parallel bytes, detecting byte-alignment commas and immediately aligning the bytes. 8b/10b encoded 10-bit words are then decoded into 8-bit data or comma bytes. Up to this point, logic has been clocked using clocks derived from the recovered line clock f_{rGHz2}, but now the recovered 32-bit events (or commas) need to be transferred to registers and logic has to be clocked by clock signal f, which belongs to a different clock domain (the one derived from $f_{refclk1}$). This clock domain "crossover" is handled using an "Elastic Buffer" [50] provided by the FPGA manufacturer within the wrapper. An Elastic Buffer is an asynchronous FIFO which is written using one clock and read using another clock. The frequencies of the two clocks are fairly similar but not exactly equal. As a result, the elastic buffer will slowly either fill up or empty out. This is avoided by defining a clock correction comma. The clock correction comma, which can be defined as a single byte, a 2-byte group, or a 4-byte group, has to be inserted into the data stream sent by the user-designed transmitter TX with certain periodicity (the exact periodicity depends on the expected worst case discrepancy between f_{GHz1} and f_{GHz2}). If the elastic buffer fills up above a given threshold, one clock correction comma is removed and not delivered to the output port. This way, the elastic buffer is suddenly emptied by the amount of one comma. On the other hand, if the elastic buffer is emptied below another given threshold, one clock-correction comma is inserted in the elastic buffer, and this comma has to be ignored at the user-designed receiver, RX in Fig. 3-2 and Fig. 3-5. The elastic buffer is thus suddenly filled by the amount of one comma. Since we were using 4-byte events, it made sense to use 4-byte clock-correction commas to avoid event de-alignment after clock-correction. However, we decided to use a one-byte clock-correction comma (BC) for lower comma traffic, which combined with our byte-alignment circuit, allows for on-the-fly event re-alignment. The use of one-byte clock correction commas had the effect of changing the alignment offset in Fig. 3-3, either incrementing it or decrementing it by '1'. The insertion and removal by the wrapper of user-defined clock-correction commas is signaled by the 3-bit signal 'clkcorr'. The user circuit RX state machine is designed to properly handle these commas.

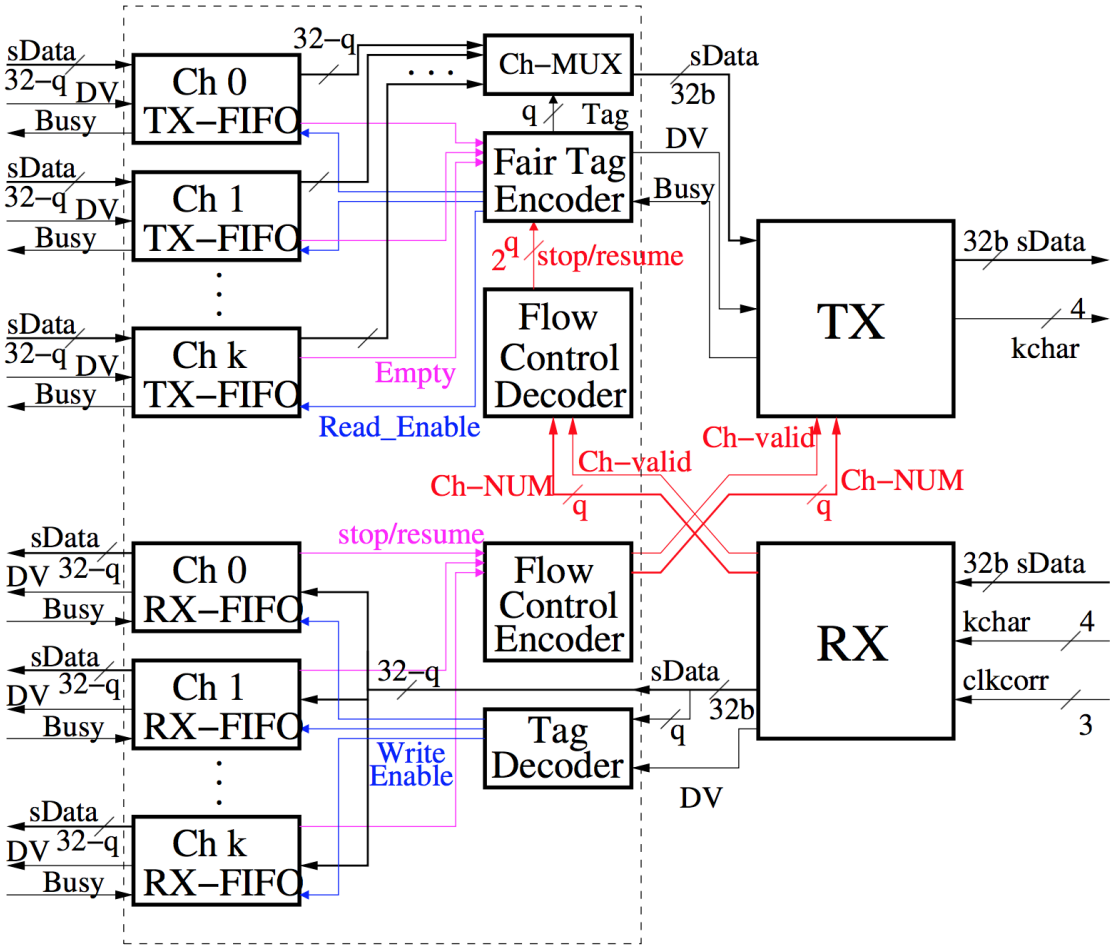


Figure 3-7: Block Diagram of Channel Multiplexing Arrange with k transmitting and k receiving channels together with flow control and tag handling blocks.

3.5 Multiple Channel Multiplexing

One SATA link with a line frequency of 3.0Gbps using 8b/10b byte encoding and transmitting 32-bit events can transmit at a rate of 75Meps (including control commas) per link direction. This event rate is fairly high compared to transmission speed through a standard parallel asynchronous AER port. For example, purely asynchronous 128×128 pixel DVS sensors have been reported to achieve speeds of almost 10Meps for 15-bit events [2]. The synchronization circuitry within the async2sync and sync2Async blocks also results in event transactions of a maximum of 6 clock cycles (if the conditions discussed in Section 3.3 are met). This would result in an event

rate of 12.5Meps. To take advantage of the available bandwidth in these SATA links, several asynchronous parallel AER channels can be multiplexed over one link. Fig. 3-7 illustrates our proposed multiplexing of $2 \times k$ AER channels over one bidirectional SATA link (k channels in each direction). The blocks shown within the broken lines in Fig. 3-2 and Fig. 3-5 are now replaced by the blocks shown within the broken lines in Fig. 3-7. If q is the smallest integer such that $2^q \geq k$, then the top q bits of the 32 sDATA bits are sacrificed to encode AER channel number within each event. Each TX-FIFO receives AER events of $32 - q$ bits and writes them into its registers, activating an output signal "Empty" when there is no data left in its registers. The "Fair Tag Encoder" FSM selects one non-empty TX-FIFO to read, following a fair selection algorithm (see below in Section IV-B). This TX-FIFO channel is selected by the Ch-MUX multiplexer block. The $32 - q$ bit data is read, the corresponding top q bits (Channel ID) are appended, and the complete 32-bit sDATA is then sent to the TX block if its "Busy" signal is non-active. On the receiver side, the "Tag Decoder" block reads the top q bits and activates the corresponding "Write-enable" signal for the destination RX-FIFO, which reads the lower $32 - q$ bits of the incoming sDATA (32 bit) word. Each RX-FIFO will be read out through its output channel.

3.5.1 Flow Control

If the readout speed at the output of an RX-FIFO is slower than the speed at which events are received, the corresponding RX-FIFO will fill up. RX-FIFOs will activate a "stop" signal if they are filled beyond a pre-set threshold (which should be lower than the FIFO's capacity), or a "resume" signal if they are emptied below a second pre-set threshold. This one-bit stop/resume signal is read by the "Flow Control Encoder" FSM, which will tell the TX block with the highest priority the channel number "Ch-NUM" whose RX-FIFO is getting close to full while activating the "Ch-valid" signal. The TX block will send a 32-bit flow control comma (CH 1C 1C 1C), with kchar = '0111', where CH is an 8-bit byte in which: (a) the 7 most significant bits encode the channel number (so that up to 128 channels can be multiplexed), and (b) the least significant bit is either '1' to signal "stop" or '0' to signal "resume".

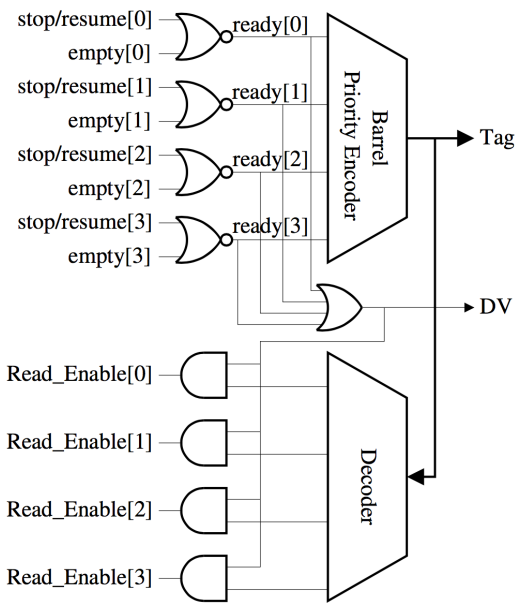


Figure 3-8: Fair Encoder Operation Diagram

This flow control comma is received by the RX block on the other side of the link, which will communicate the channel stop/resume command to the "Flow Control Decoder" FSM. This FSM will then decode the saturating channel's ID and set the corresponding stop/resume signal (one of the 2q lines) for the "Fair Tag Encoder", which will in turn enable/disable the "Read Enable" signal for the channel, so that the corresponding channel TX-FIFO will stop/resume accepting new input events.

Note that this flow control scheme is very similar to the single-channel scheme explained in Section 3.3, except that here the flow control comma uses all 8 bits of the first byte, and the upper q bits of the events are sacrificed to encode channel number.

3.5.2 Fair Tag Encoding Operation

Fig. 3-8 shows a simplified diagram of the "Fair Tag Encoder" block. Channel signals 'stop/resume' and 'empty' are OR-ed to generate 'ready' signals. These 'ready' signals are fed to a "Barrel Priority Encoder". This block is a priority encoder whose priority preference is circularly shifted one position each clock cycle, as illustrated in Fig. 3-

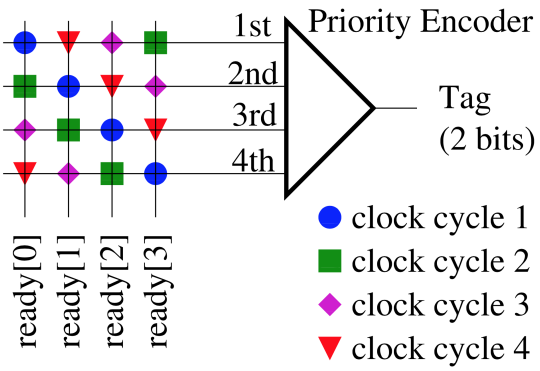


Figure 3-9: Schematic Operational Diagram of Barrel Priority Encoder

9. The "Barrel Priority Encoder" generates the corresponding output channel Tag, which is used by a Decoder circuit to generate 'Read Enable' signals.

3.6 Experimental Results

In the experimental measurements reported here, we use Spartan-6 GTP interfaces operating at 3.0Gbps with error-free transmission. This data rate is very close to their maximum data rate limit of 3.2Gbps. This requires careful PCB design and component choices. The high-speed traces on the PCB were designed using industry-standard techniques. The PCB manufacturer supplied track width and spacing based on the proposed board stack-up and required impedance. The Cadence Allegro PCB tools were set up to use these parameters. The differential pairs were automatically length matched, and all bends were chamfered, and vias avoided where possible. The high-speed tracks were routed on the outer layers of the PCB with a ground plane beneath. Standard surface mount SATA connectors were used on the PCB. During board commissioning, the drive strength of the FPGA differential drivers was adjusted to ensure adequate noise margin on the links.

3.6.1 Test of Accelerated Sync2Async and Async2Sync Scheme

In a preliminary characterization, we tested first the correct operation of the accelerated Async2Sync and Sync2Async scheme presented in Section 3.3. Fig. 3-10(a)

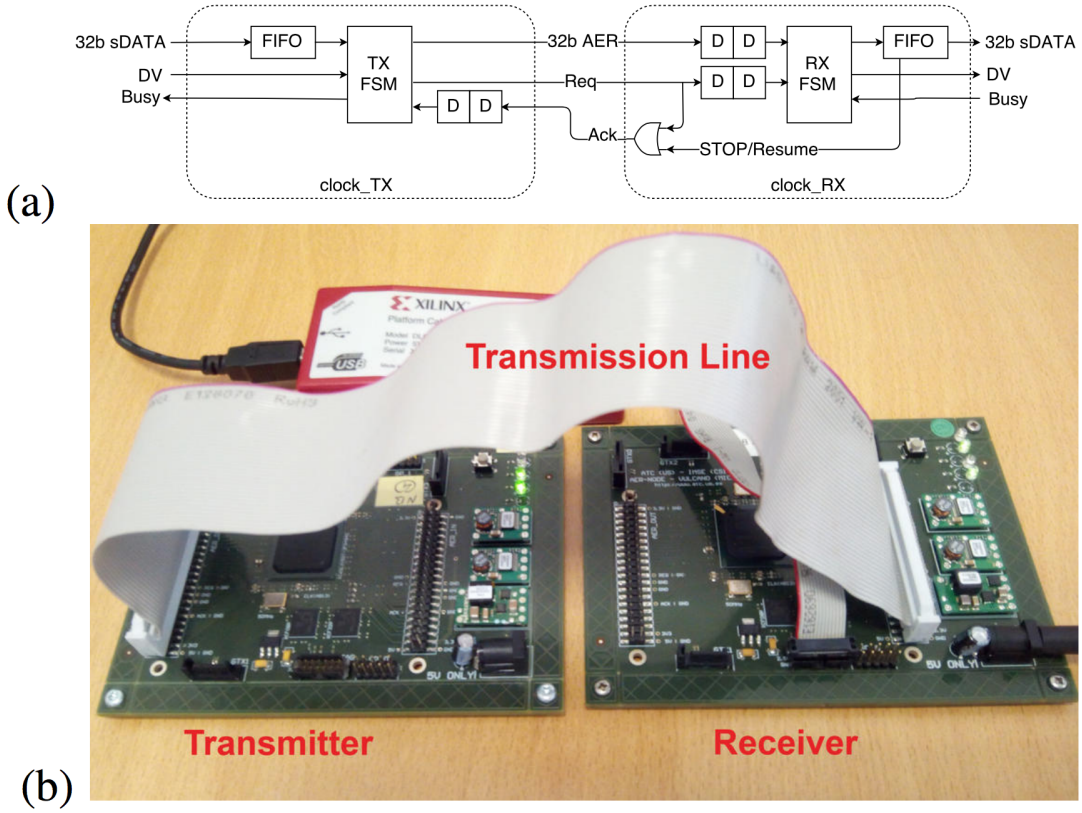


Figure 3-10: Separate setup to test and characterize the accelerated Sync2Async and Async2Sync scheme. (a) Schematic diagram, and (b) Experimental setup.

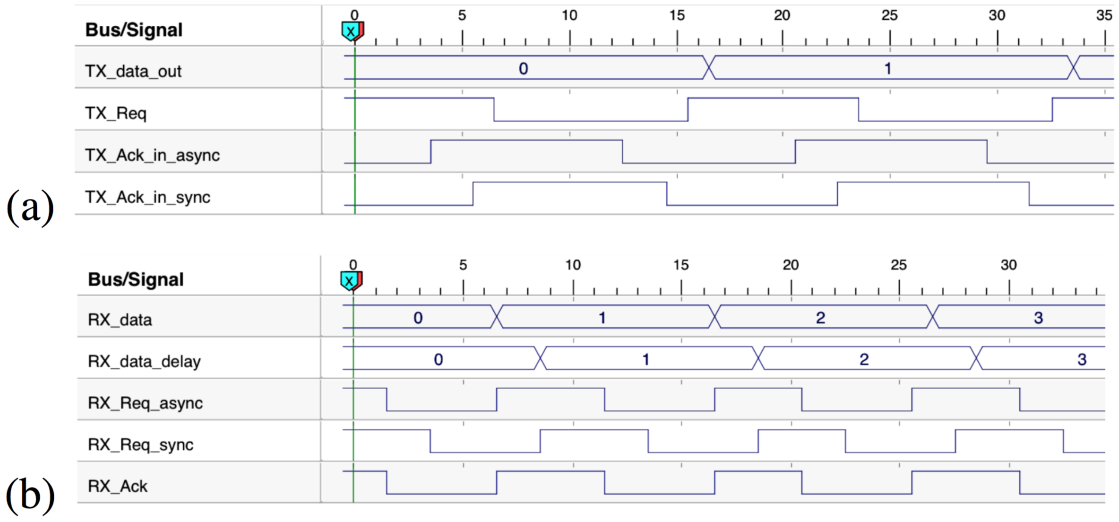


Figure 3-11: ChipScope captured signals at (a) TX circuit with 250MHz clock and (b) RX circuit with 143MHz clock frequency.

shows the schematic diagram of the setup used for this, with the TX on the left and the RX on the right sides, each with its own independent clock. Let us call their clock periods P_{TX} and P_{RX} respectively. At TX the 32-bit AER data is set one clock cycle before Req. Ack returns (if STOP/resume signal is low) without passing through any synchronization nor state machine at the receiver. If wire delays are negligible, the TX requires $6P_{TX}$ to perform one full data transfer: 1st to write data, 2nd to write Req, 3rd to capture Ack, 4th, and 5th to capture the synchronized version of Ack by the TX FSM, and 6th to allow the FSM to write the new data. If there are delays in the wires, pads, and OR gate, then one event transmission requires $6P_{TX} + 4\tau_{line}$ (where $2\tau_{line}$ includes one full round: two physical wires and connectors, four pads, and the OR gate). On the RX side, the circuit needs $3P_{RX}$ to capture one event. If RX clock is faster than TX clock ($P_{TX} > P_{RX}$) there are no communication problems. However, if TX clock is faster, we have to guarantee that $6P_{TX} + 4\tau_{line} > 3P_{RX}$. The worst case is when τ_{line} is negligible, which results in $2P_{TX} > P_{RX}$ (or $2f_{RX} > f_{TX}$). Consequently, if TX clock frequency is not more than twice the RX clock frequency, correct communication is guaranteed, independent of the physical delays of lines, pads, connectors, etc.

To verify this, we assembled the experimental setup shown in Fig. 3-10(b) with intentional long external wires. Two AER-Node boards [6] were used, interconnected through their parallel ports with a relatively long parallel bus ribbon cable. The TX circuit was clocked at 250MHz while the RX was clocked at 143MHz. This setup showed error-free transmission tested over several days. Fig. 3-11(a) shows signals Data, Req, Ack at the pad, and Ack after synchronizers, captured inside the TX FPGA using ChipScope. We can see that one event cycle transmission requires 17 TX clock cycles (68ns), although sometimes it would require 16. The measured average was 16.98 cycles (67.93ns). Since we estimated this delay as $6P_{TX} + 4\tau_{line}$, it results in $\tau_{line} = 10.98ns$ (equivalent to 2.75 TX clock cycles). Fig. 3-11(b) shows signals Data at the pads, after synchronizers, Req at the pad, after synchronizers, and Ack at the pad, captured inside the RX FPGA using ChipScope. We can see one event cycle transmission of 10 RX clock cycles (69.93ns) and another one of 9 RX clock cycles (62.94ns). The measured average was 9.71 RX clock cycles (67.93ns). Therefore, this accelerated scheme setup was able to transmit at an average of 67.93ns per event, or equivalently, 14.72Meps. By using the non-accelerated scheme, the average event transmission rate was 9.75Meps.

In the experiments that follow, the Async2Sync and Sync2Async interfaces are completely inside the FPGAs thus minimizing τ_{line}. The measured results shown next also demonstrate error-free transmissions.

3.6.2 Serial Link Characterization

Fig. 3-12 and Fig. 3-13 show two typical setups in which the proposed bidirectional serial link was used. The figures show two separate AER retina sensors, each connected to the AER-Node Board [6] by a parallel AER connector (using a custom adapting PCB that makes it possible to connect up to 4 AER parallel ports). The two retinas communicate with the Spartan6 FPGA on the AER-Node Board, which in turn communicates with one 48-chip SpiNNaker Board [1] through serial SATA. The SpiNNaker board receives events from the two retinas' AER ports, processes them and sends the resulting event flow back to the AER-Node Board via the same

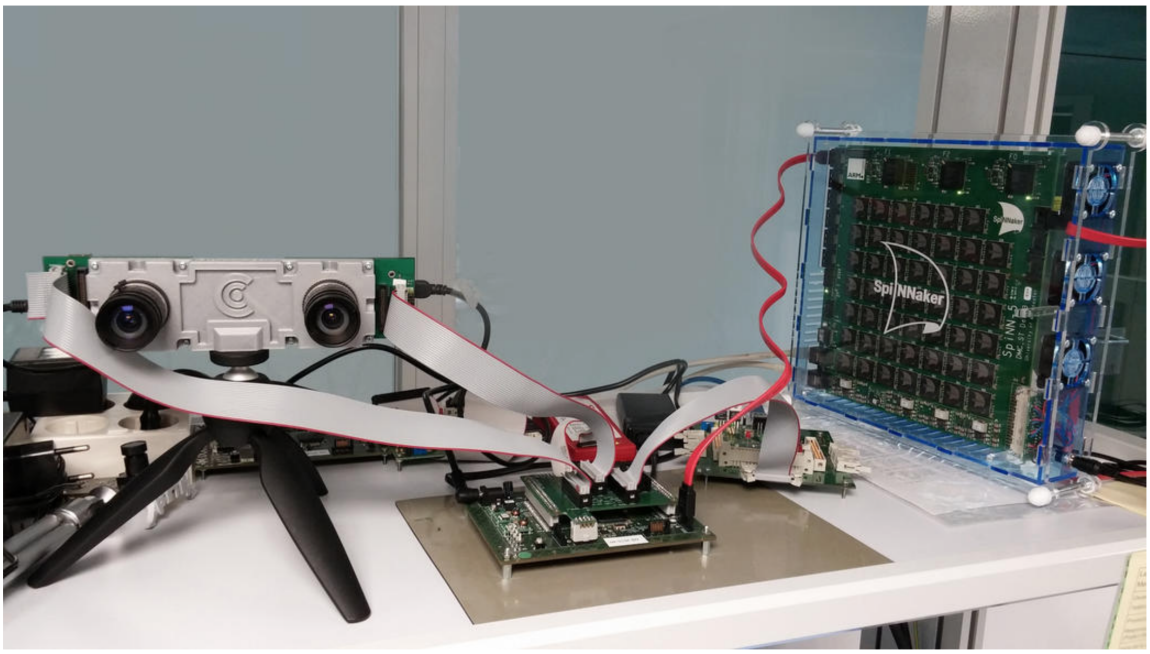


Figure 3-12: Example setup with two ATIS retinas [5] connected via AER parallel ports to the AER-Node board [6], which connects via SATA to a 48-chip SpiNNaker Board [1].

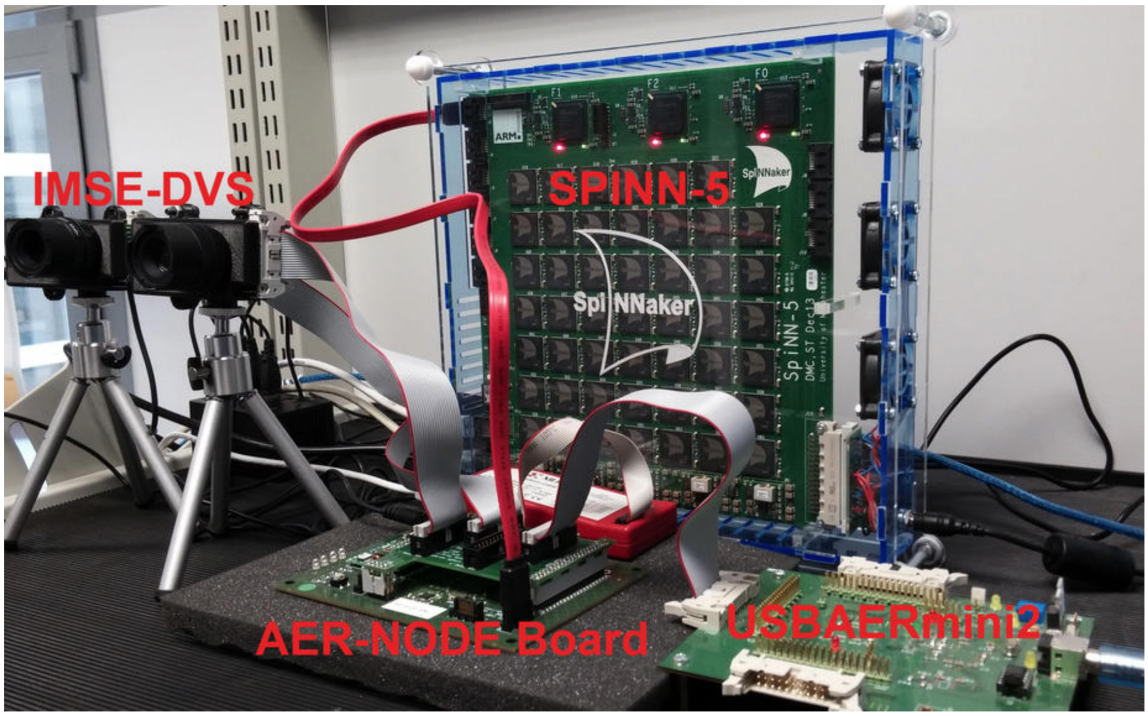


Figure 3-13: Example setup with two DVS [2] retinas connected via AER parallel ports to the AER-Node board [6], which connects via SATA to a 48-chip SpiNNaker Board [1].

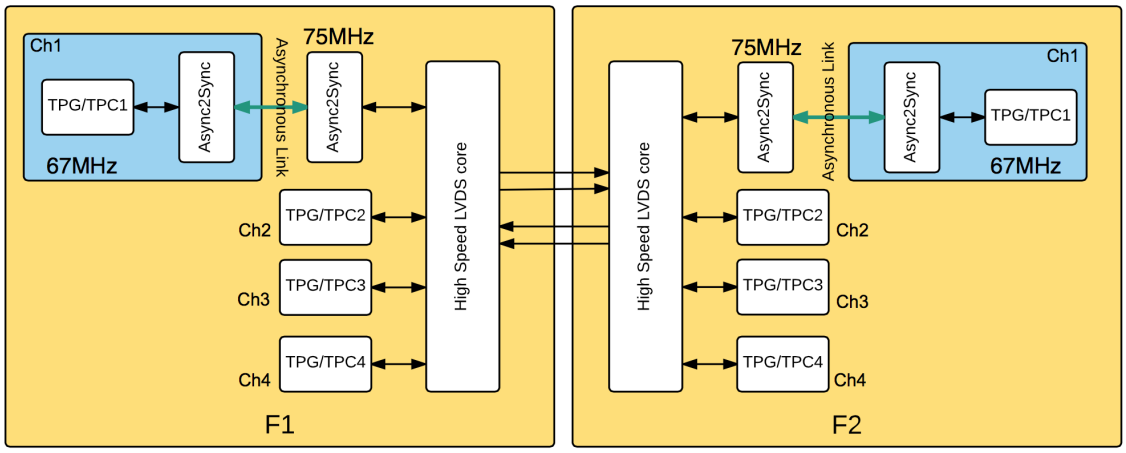


Figure 3-14: Simplified Diagram illustrating the Experimental Configuration inside the two Spartan6 FPGAs

SATA wire. The AER-Node Board sends the results through another parallel AER port to a USBAERmini2 Board [3] which communicates with a host computer by USB to display the results in real time.

The proposed communication scheme was experimentally characterized using a pair of Spartan6 FPGAs located on two different (SpiNNaker) PCBs. Each Spartan6 (XC6SLX45T-3) FPGA used its own 150MHz Xtal oscillator. To test the multiplexed link at maximum throughput (thereby, forcing flow control), we used one GTP port on each FPGA, as illustrated schematically in Fig. 3-14. Each FPGA used its local 150MHz reference clock plus an additional 67MHz clock, so each FPGA included two separate clock domains. The setup was configured with four separate bidirectional AER Channels (3 synchronous and one asynchronous) multiplexed over the SATA link. For this, we used a "Test Pattern Generator" (TPG) and "Test Pattern Checker" (TPC) transmitter/receiver pair. The TPG provides a known sequence of patterns, while the TPC checks and counts event errors in that sequence and computes the effective event rate received (excluding all control commas). In each FPGA, three TPG/TPC pairs were clocked with the same clock as the Transceiver/Multiplexing core discussed in Fig. 3-2, Fig. 3-5 and Fig. 3-7. This is a clock running at f = 75MHz, derived from the external 150MHz Xtal reference oscillator. Each of the three synchronous TPGs could therefore provide an event rate of up to 75Meps (one

Table 3.2: Experimental characterization test for fast 6-cycle event transaction synchronization scheme on channel 1

Channel#	Event Rate (Meps)	Link utilization(%)	Error Ratio
Channel1	11.14	14.86	0
Channel2	26.29	35.05	0
Channel3	18.75	25.00	0
Channel4	18.75	25.00	0
Total	74.93	99.90	0

Table 3.3: Experimental characterization test for slower 12-cycle event transaction synchronization scheme on channel 1

Channel#	Event Rate (Meps)	Link utilization(%)	Error Ratio
Channel1	6.43	8.57	0
Channel2	31.00	41.33	0
Channel3	18.75	25.00	0
Channel4	18.75	25.00	0
Total	74.93	99.90	0

per clock cycle). The 4th TPG/TPC pair was clocked with the additional 67MHz clock and interfaced through a pair of async2sync blocks. The LVDS line rate was set at 3.0Gbps.

We tested the setup using the two synchronization schemes discussed in Section 3.3 and Fig. 3-4: the faster one requiring on average six clock cycles per event transaction, and the slower one requiring on average 12. Table 3.2 summarizes the results for the 6-cycle case after testing the setup for 65 hours. None of the channels detected a single error in the transmission. The link bandwidth (75Meps) was shared by the four Channels as indicated in Table 3.2, covering effective data events plus control commas (for flow control, clock correction, or idle commas). The TPGs clocked at 75MHz attempted to deliver data at one event per clock cycle, but were slowed down by the corresponding "Fair State Encoder" whenever the SATA link bandwidth was reached. The TPG clocked at 67MHz (Channel 1) could only deliver events at a much lower rate because of the synchronization delays. This explains why the effective event rate for this link dropped to 11.14Meps. The rest of the bandwidth was split up between the other Channels as shown in Table 3.2. Note that

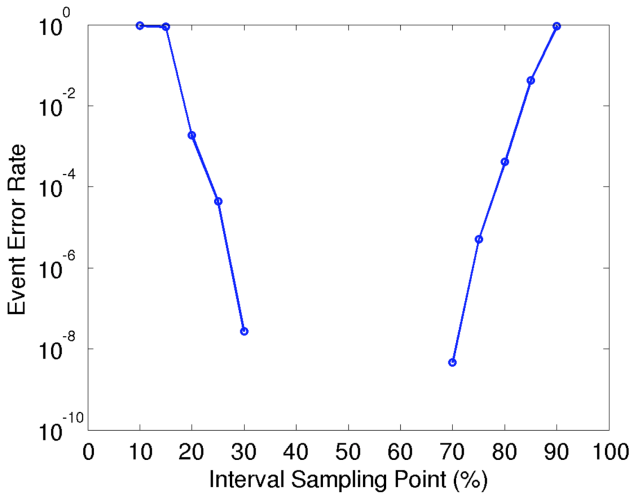


Figure 3-15: Measured Event Error Rate while sweeping Interval Sampling Point

Channel2 ends up transmitting at a higher rate than Channels3-4. This is because of the priority encoder design in Fig. 3-8 and Fig. 3-9, which assigns to each Channel different priorities every clock cycle (to prevent one pAER transmitter from blocking others). Since Channels 2 to 4 always have data ready to send, they will use 25% of the time (when they have the highest priority) to send their data. When highest priority is with Channel1, sometimes there is no data ready to send. In this case, Channel2 has second priority and will use this time for its data.

Total data bandwidth was thus 74.93Meps (99.90% of link bandwidth). The remaining 0.07Meps bandwidth (0.10%) was used by control commas. Table 3.3 shows the same results, but for the case of using the slower 12-cycle synchronization scheme. In this case, Channel 1 achieved a lower throughput, but the total link bandwidth was kept the same. Note that Channel1 event rate in Table 3.3 is slightly faster than half of that in Table 3.2, which might be surprising because we are expecting the fast scheme to require between 5-to-6 clock cycles and the slow scheme between 10-to-12. When using the fast scheme, throughput is limited by the TX clock at 67MHz. However, for the slow scheme, the delays depend on both the TX and RX clocks, and the effective clock cycle is somewhere between 67MHz and 75MHz, resulting in an event transmission rate slightly better than half of the fast one.

The FPGA GTP receivers made it possible to tune the voltage sampling point

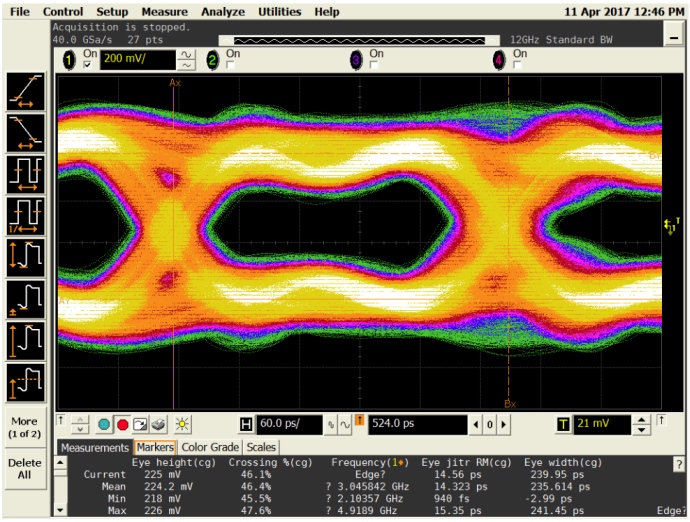


Figure 3-16: Measured Eye Diagram on the LVDS lanes operating at 3.0Gbps.

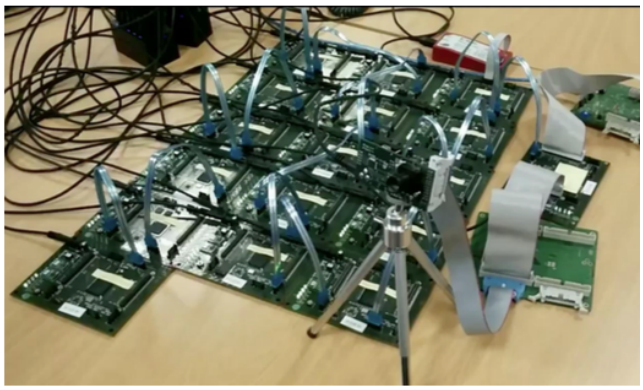
of the physical LVDS line to compensate for potential asymmetries. This sampling point is expressed as a percentage of the full range, the central point being 50%. We tested the event error rate of the link as a function of the interval sampling point. The results are shown in Fig. 3-15, where it can be seen that event error rate was null between sampling points 0.35% and 0.65%. Outside this range, the event error rate increased exponentially.

Fig. 3-16 shows the eye diagram measured using an Agilent DSO81304B oscilloscope with 12GHz bandwidth soldered probes. Eye opening was 235ps width times 224mV height, with an average RMS jitter of 14.3ps. This confirms a safe enough margin on the physical design side for the transmission speed of 3Gbps (333ps per bit). We can see in the figure that transmission of one bit requires an average of 328ps, which yields an average transmission frequency of 3.046GHz. The figure also shows that the differential amplitude of the physical LVDS signal has an average of about 700mV peak to peak.

3.6.3 Application Example

Fig. 3-17(a) illustrates a multi-FPGA multi-PCB application example of a neuromorphic system that extensively exploits the presented LVDS interface protocol technique. The setup shows 17 Spartan6-based AER-Node Board PCBs [6] which receive

(a)



(b)

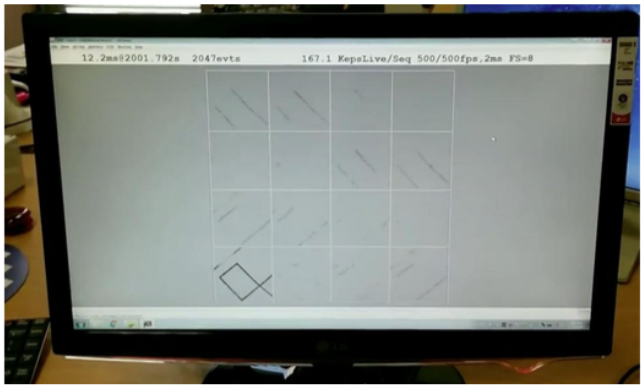


Figure 3-17: Example application of neuromorphic event-driven sensing and computing setup including an event-driven Dynamic Vision Sensor and array of event-driven filtering blocks emulating the V1 layer of the vertebrate visual system. (a) Physical setup using 17 Spartan6 AER-Node Boards [6] intercommunicated through our SATA serial protocol, one event-driven vision camera [2], and one USBAERmini2 computer interfacing PCB [3] to monitor computations in real time on a monitor screen, shown in (b).

real-time events from an event-driven spiking retina Dynamic Vision Sensor [2]. The sensor event flow is distributed to a mesh of event-driven convolution filters [40] that emulate the operation of the vertebrate V1 visual cortex. These convolution filters are implemented on the Spartan6 FPGAs, and the event traffic is distributed using the SATA LVDS links. The output event flow of each convolution filter is then routed back through the same SATA LVDS links and concentrated on one of the AER-Node Boards, which converts the input and a selected number of outputs to parallel AER, which is then interfaced to a PC through a USBAERmini2 PCB [3], to monitor in real time the activity of the selected V1 filters on a screen, as shown in Fig. 3-17(b), using jAER [4].

Typical techniques for mapping generic spiking networks onto modular hardware exploit the mapping of computational architectures to 2D meshes [35], [40], where unit elements are connected to nearest neighbors. Each unit element includes a processing module and a router. Each router contains its own routing table, and the set of all routing tables defines how the original computational architecture has been mapped onto the 2D array. In this approach, physical links only exist between neighbors. This may result in congestions if, for example, two remote unit elements have to maintain a high event data rate between them, because this event traffic would use time of all the routers in the path. However, with the proposed technique of multiplexing multiple AER paths on the same physical SATA wire, it is possible to establish direct routes between remote unit elements without necessarily going through all the routers within the 2D mesh path.

3.7 Conclusion

We have presented a method for multiplexing multiple asynchronous and/or synchronous Address-Event-Representation channels over a physical bidirectional inter-FPGA LVDS link. The scheme allows for the separate, independent flow control of each AER channel and includes proper byte alignment control for the serial communication, together with clock correction techniques for compensating clock drifts

between the reference clocks of the FPGAs. Experimental results using the LVDS links of two Spartan6 FPGAs on separate boards demonstrated the correct operation of the link. Exhaustive tests were carried out in hardware, showing error-free transmissions over extended time periods while communicating events at the full bandwidth of the physical links.

Although the lowest data transmission layers are well known and widely used (Xilinx IP wrapper), our contribution in the field of biology-inspired computing is the setup of heterogeneous architectures able to combine various custom processing elements (like concurrent ARM-based SpiNNaker platform, massively parallel FPGA-based emulators of spiking neurons) connected to each other via different asynchronous interfaces (2-of-7 parallel bus, pAER 16-bit parallel bus, bi-directional LVDS serial links) and driven by multiple event-based neuromorphic sources of real sensory signals (such as artificial Retinas or Cochleas). Even though computing elements (CPU's and FPGA's) and transmission links operate asynchronously and at different speeds, we have proven that they are able to cooperate and the transmission can be error-free even when reaching physical data rate limits. One particular concern and contribution in this work was sharing the bandwidth of single LVDS channels by various agents (multiple transmitters and receivers). We demonstrate error-free transmission running at almost maximum possible speed sharing the bandwidth of single bit-serial channels among numerous synchronous and asynchronous elements running at different rates. Priority encoding and flow-control are critical for avoiding buffer overflow when control and data events appear concurrently. Illustrations of multiple and heterogeneous neuromorphic setups are provided.

Chapter 4

Fast Pipeline 128×128 Pixel Spiking Convolution Core for Event-Driven Vision Processing in FPGAs

This work has been published in:

A. Yousefzadeh, T. Serrano-Gotarredona and B. Linares-Barranco, "Fast Pipeline 128×128 *pixel spiking convolution core for event-driven vision processing in FPGAs," 2015 International Conference on Event-based Control, Communication, and Signal Processing (EBCCSP), Krakow, 2015, pp. 1-8.*

4.1 Abstract

This chapter describes a digital implementation of a parallel and pipelined Spiking Convolutional Core to be used in spiking convolutional neural network (S-ConvNet) for processing spikes in an event-driven system. Event-driven vision systems use typically as sensor some bioinspired spiking device, such as the famous Dynamic Vision Sensor (DVS). DVS cameras generate spikes related to changes in light intensity. In this chapter, we present a 2D convolution event-driven processing core with 128×128 pixels. S-ConvNet is an Event-Driven processing method to extract event features from an input event flow. The nature of spiking systems is highly parallel, in general.

Therefore, S-ConvNet processors can benefit from the parallelism offered by Field Programmable Gate Arrays (FPGAs) to accelerate the operation. Using three stages of pipeline and a parallel structure results in updating the state of a 128 neuron row in just 12ns.

Keywords: Spiking Convolutional Neural Networks, DVS, Artificial Retina, FPGA, Parallel Processing

4.2 Introduction

After generating and sending spikes in a retina chip, the next step is to process the spikes and extract the desired features. Work on AER systems started around twenty years ago [24], but AER processing with generic feature extraction hardware is more recent [51, 52, 40, 3, 19, 53].

Some simple event-driven processing can be implemented in software [4, 54]. However, software implementations suffer from high latencies due to processor resource sharing between all neurons. For event-driven processing, some mixed analog-digital chips [55] have been designed that achieved good results but they suffered from a high mismatch in the analog circuitry, which required expensive in-pixel calibration. Fully digital ASICs were also designed [51, 52] but only one ConvNet Feature Map could be implemented in one low-cost chip.

In this chapter, we introduce a fully digital implementation of a spiking convolutional event-driven core that can be implemented in commercial FPGAs. We present a pipelined scheme capable of updating 128 synaptic connections in 12ns. This improves with respect to previously reported FPGA convolutional event-driven cores where 121 synaptic updates where performed in 3us [40], or 84 synaptic updates in 10ns [19]. In the next Section, we will describe the spiking convolutional neural network concept briefly. Then the proposed core will be presented, and finally, we will provide the implementation results.

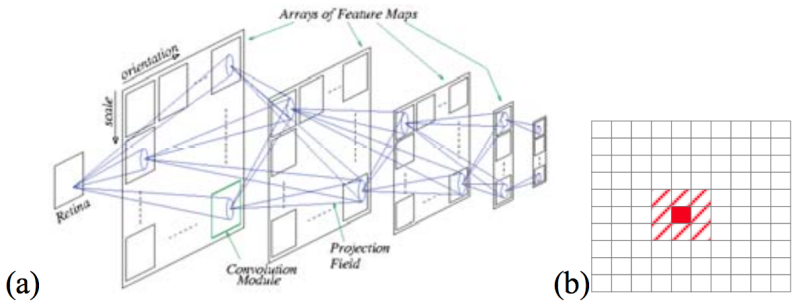


Figure 4-1: (a) Illustration of event-driven convolutional computation concept. (b) Generic ConvNet with several layers, each with several Feature Maps.

4.3 Convolutional Neural Network Concept

In conventional frame-based image processing, one typical strategy is using kernel convolutions to extract image features, combine them progressively and perform object recognition. To exploit this concept in neural networks, the Convolutional Neural Network (ConvNet) paradigm was developed to define how to learn kernel weights (synaptic weights) [56]. If we use a kernel size of $M \times N$, each neuron in a layer connects to the next layer through $M \times N$ synapses. The distribution of synaptic weights is the same for all neurons in previous layers. This is also known as "weight sharing". Fig. 4-1(a) shows a generic ConvNet structure with multiple layers, each layer with several "Feature Maps" (FM). Each Feature computes several convolutions, depending on the origins of the events. Each FM is a 2D grid arrangement of neurons receiving spike events from neurons of the previous layer FMs. A neuron in a FM sends its spikes to a "projection field" of neurons in a receiving FM in the next layer. This is done by assigning kernel weights to synapses and is illustrated in Fig. 4-1(b). Assume that the 10×10 grid is an FM which received a spike from a source neuron in the coordinate in the red position. If the kernel size is 3×3, the incoming spike will convey to a 3×3 square through synapses. This square is the "projection field" of the source neuron. The weights of the synapses are extracted from the kernel weights. For more details refer to [51, 52, 55].

As an illustration of this event-driven convolutional processing, let us consider the case illustrated in Fig. 4-2. Fig. 4-2(a) shows a 124ms histogram obtained by

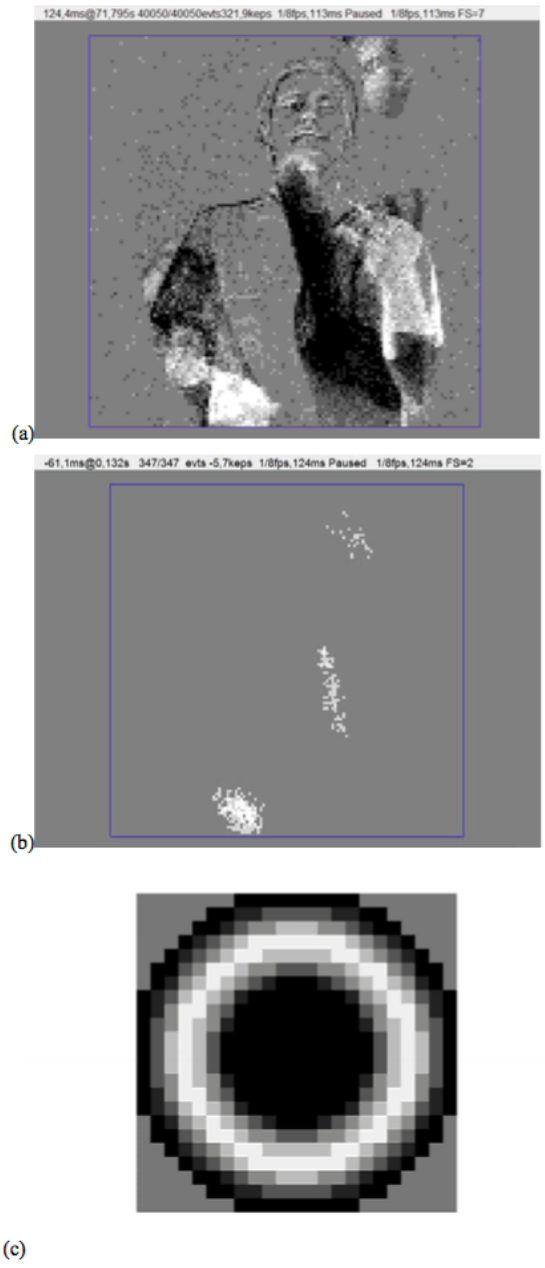


Figure 4-2: Event-Driven 2D Convolution Processing. (a) 124ms histogram from a DVS output. (b) Output of reported 2D convolution processing core programmed with the kernel shown in (c).

collecting the events from a DVS retina while observing a person juggling with three balls. The DVS retina has 128x128 pixel resolution. Each ball has a diameter of about 16 pixels in Fig. 4-2(a). The retina can generate events at a rate of up to 10Meps (million events per seconds). In this particular recording, the event rate generated by the retina is 322keps (thousand events per second). If every event generated by a retina pixel is sent to our 128×128 pixel convolution processing core programmed with the 23×23 pixel convolution kernel in Fig. 4-2(c), as illustrated in Fig. 4-1, the convolution core output is as shown in Fig. 4-2(b). This is because the convolution kernel in Fig. 4-2(c) is tuned to detect circles of 16-pixel diameter: pixels on the diameter contribute positively to the center of the circle, while pixels in the central region or slightly beyond the 16-pixel diameter contribute negatively to the center. This way, the output of the event-driven convolution processing, as shown in Fig. 4-2(b), highlights the centers of the 16-pixel diameter balls.

Fig. 4-3 helps to better understand the intuition behind this event-driven convolutional processing. The left side in Fig. 4-3 shows a solid ball moving in the real world. The central plane in Fig. 4-3 represents the pixels of a DVS sensor, which detect illumination changes. Thus, only the pixels on the peripheral circumference will become active and send one or more spikes.

When a retina pixel sends a spike to the convolution core on the right of Fig. 4-3, it will send spikes to the projection field defined by the convolution kernel in Fig. 4-2(c). This projection field sends a positive contribution to the pixels at the circumference of diameter equal to 16 pixels, while the contribution is negative inside and slightly outside that circumference. When adding up the contributions of all projection fields of the retina active pixels, there will be a net positive contribution in the center of the original circumference on the convolution core plane, signaling the center of the circumference of the expected size. This is what is shown in Fig. 4-2(b).

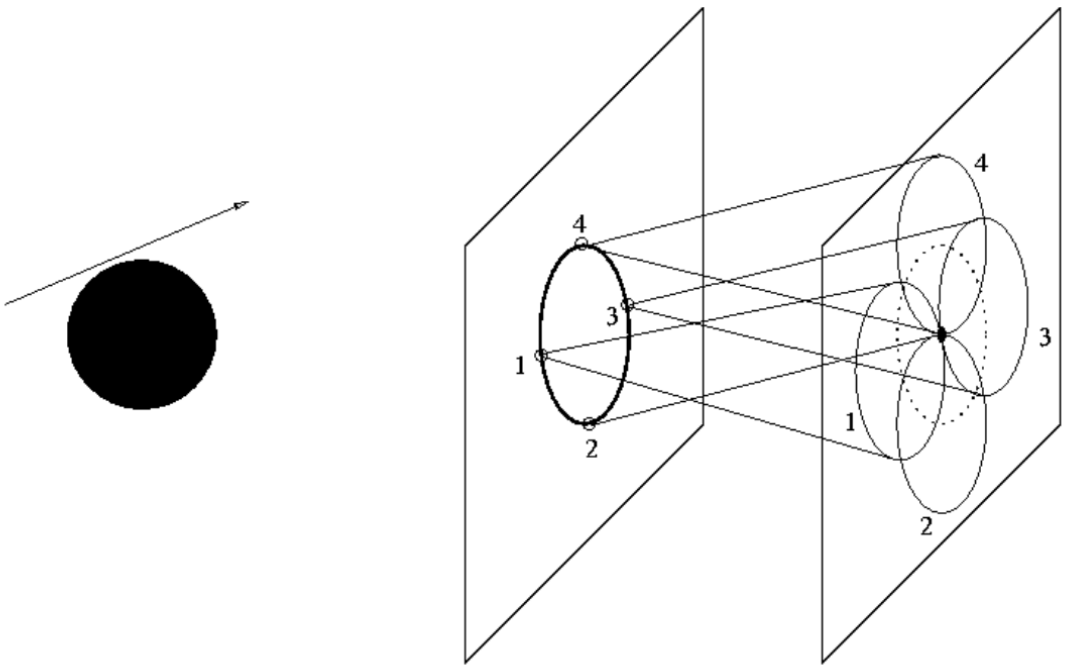


Figure 4-3: Event-Driven 2D Convolution Processing when observing moving balls. Left: ball moving in reality. Center: output provided by a DVS retina, where the pixels on the periphery of the ball generate events, as those are the pixels detecting changes in light. Pixels 1, 2, 3, and 4 on this periphery project the convolution kernel on a 128×128 pixel array inside the convolution processing core. Each retina pixel contributes positively on the projecting 16-pixel diameter circumference. The positive contributions of all pixels at the retina plane add up at the center of the circumference in the convolution core plane, signaling the presence of a 16-pixel diameter circle.

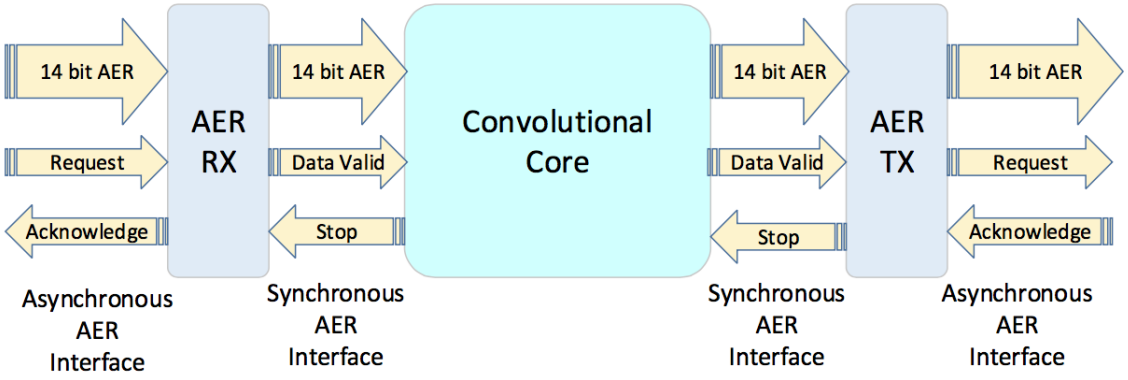


Figure 4-4: Convolutional core interfaces.

4.4 Proposed Event-Driven Convolutional Core

The implementation of the convolutional core presented in this work contains a fast parallel and pipelined hardware structure for event processing. The retina provides completely asynchronous events. In the FPGA, a complete asynchronous design is not efficient. In this chapter, a Globally Asynchronous Locally Synchronous (GALS) structure is designed to share some resources and take advantage from a pipeline and parallel design principles. With this approach, we obtained a high rate of event processing, while using an optimum number of cells in the FPGA. In this chapter a simple leaky integrate and fire neuron model with instant synapses [57] has been used.

Fig. 4-4 illustrates the connections between the convolutional core and the AER modules for receiving and transmitting events through asynchronous parallel AER interfacing. The core contains three main modules "Convolution Core", "AER RX", and "AER TX". The AER receiver and AER transmitter are designed to communicate with asynchronous AER protocol PCBs [3] and change the event flow into a fast synchronous protocol to communicate with the convolution core. Both asynchronous and synchronous protocols include flow control. The synchronous protocol can send one event per clock cycle, while the asynchronous protocol within the FPGA is slower.

The convolution core itself has three major blocks that work in parallel. The first block manages input events and updates the neurons states by doing event-driven

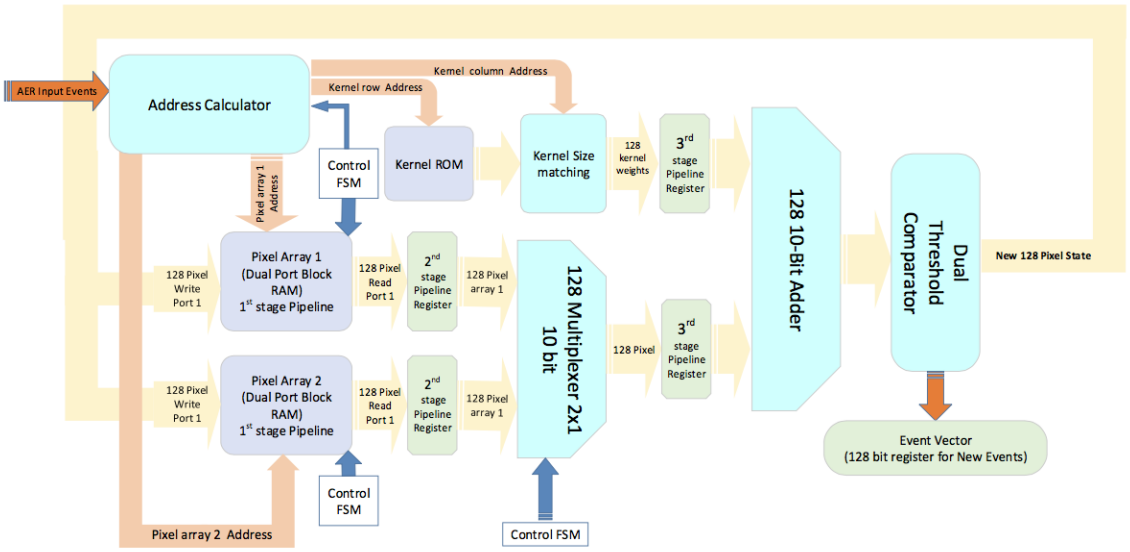


Figure 4-5: Data flow diagram of the input event processing block

convolutions. The second block is in charge of applying a forgetting rate (leakage) to the neurons, and the last block is the block for managing generated events and make them ready to be sent out of the core.

4.4.1 Input event processing block

Fig. 4-5 illustrates schematically the data flow diagram of the input event processing part. The convolutional core uses a pipelined parallel scheme to convolve one row of a kernel to one row of pixels (128 pixels in our case) in one clock cycle.

Pixel arrays hold their neuron state in dual-port block RAMs of the FPGA. Fig. 4-5 illustrates the use of one of the ports. The second port is used for the forgetting logic part and will be explained next. The membrane voltage of the neurons is saved in pixel arrays with 10 bits per pixel. The 2's complement scheme has been used to store signed numbers. In this work, we use a 128×128 pixel arrangement for the convolutional core. Each pixel state is 10-bit, and the core reads one line of pixels at the same time. Therefore, we need to use a 1280-bit bus. We used Xilinx Block RAMs that are fully synchronous. This means they need one clock cycle for reading and providing data. That is the reason for locating the first stage of the pipeline inside the pixel arrays.

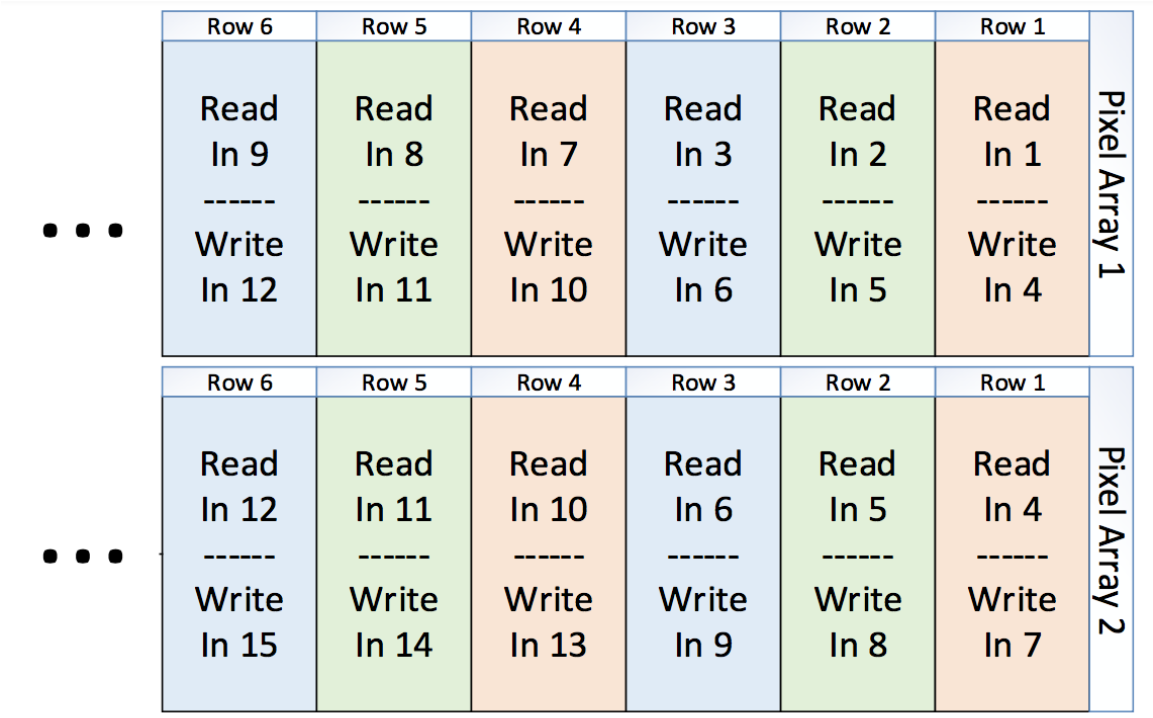


Figure 4-6: Time sequence for reading and writing from the pixel arrays. It starts by reading the first row of pixel array 1 in the first clock cycle and shows the operations until the 15th clock cycle

The process of updating one row of pixels consists of 3 steps: reading a row of pixels from the block RAM, adding the proper row of the kernel and comparing the results against a threshold. Finally, the corresponding row of pixels should be updated in the block RAM, which also needs an additional clock cycle for writing. For this purpose, it requires at least two clock cycles for reading and writing. To reach the speed of one row per clock cycle, two independent pixel arrays are used in parallel, each one containing half of the pixels. The even rows of pixels are in pixel array 1, and the odd rows are in the pixel array 2. The two modules of block RAMs let the core read a row when it is writing in the previous row of the other RAM.

Fig. 4-6 shows the time sequence for reading and writing from the pixel arrays. The first, second and third clock cycles are spent for reading from pixel array 1 to fill up the three pipeline stages. In the fourth clock cycle, the first row of the pixel array 2 will be read and the first row of the pixel array 1 will be written. In the following

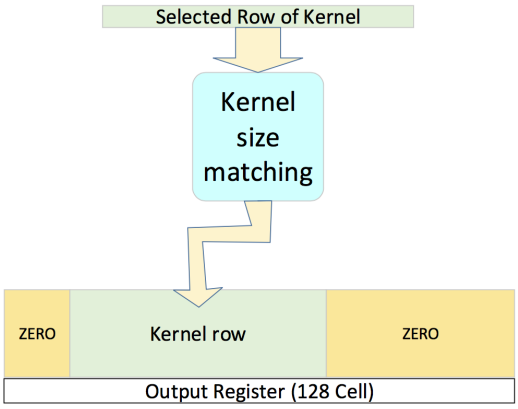


Figure 4-7: Input and output of kernel size matching logic block

clock cycles, one read and one write operation will be done in one clock cycle until doing the full convolution.

The address calculator in Fig. 4-5 is a logic block that defines the addresses of the rows which should be read and written in the pixel arrays and kernel ROM. It calculates the proper addresses based on the input event address and its current state.

Another part in Fig. 4-5 is the "kernel ROM". For the kernel, the number of bits allocated to each weight is 6 bits including sign, based on the 2's complement scheme. Normally, the kernel size is smaller than the pixel array size. Therefore, a logic is designed to find out the columns of pixels that should be added with the kernel. For this purpose, the "Kernel size matching" logic, puts the kernel row in the proper columns of an empty 128 cell register, while the other cells stay at zero. Each cell contains 6 bits. Fig. 4-7 illustrates the output of this module. The adder simply contains 128 10-bit adder blocks that add the 128 pixels to the kernel weights which are put in proper columns.

After adding the kernel to the pixels values, the threshold logic block compares the values of each pixel to a positive and a negative threshold. This block has two outputs, the new pixels values, and the event vector register.

New pixels values are the result of adding the kernel to the previous pixels values, and the threshold logic puts the reset value for the pixels that exceed the threshold. In neural network terminology, it means that the neuron fires and generates a new

spike and its state goes back to reset. The new pixel value is written in the same row of the pixel array to update the row.

Another output of the threshold logic block is the event vector. In this version of the core, negative events are not saved. So, if a pixel value goes below its negative threshold, it is reset to zero, but no new events are generated. However, if the pixel value goes above its positive threshold, it is reset to zero, and in the 128-bit event vector register, a flag related to the place of this pixel will be set to ON. Another logic block that manages and sends generated events uses this vector as input.

The number of pixel rows that should be added with kernel rows depends on the size of the kernel and the address of the incoming event. There is a control finite state machine in this part of the core that controls the flow of data and asserts the control signals (such as read and write in the pixel arrays) and selects the proper input for the multiplexer. It also controls the "address calculator" logic block and the stop signal for the synchronous interface to manage flow control. This logic block should be aware of parameters like kernel size, negative and positive thresholds and reset value of the pixels.

4.4.2 Forgetting logic block

Another critical part of the convolution core is the logic block in charge of applying leakage to the neurons. Fig. 4-8 illustrates the data flow diagram for this block.

In this block, the core uses the second port of the pixel arrays RAM. The pipeline stages, multiplexer, and address calculator are almost the same as in the previous part. When the forgetting logic and convolution logic blocks want to write in the same row of the pixel arrays, there is a conflict. For addressing this situation, a collision detector logic block is designed to detect this condition and notify the control logic. After catching a collision, the pipeline stage should become empty, and the forgetting logic has to start from the previous three rows. To minimize the number of collisions, a "forgetting process" begins from the end of the pixel array and proceeds towards the first row, while the convolution logic reads and writes in the reverse direction. Using this strategy minimizes collisions, as the maximum collision happening for one

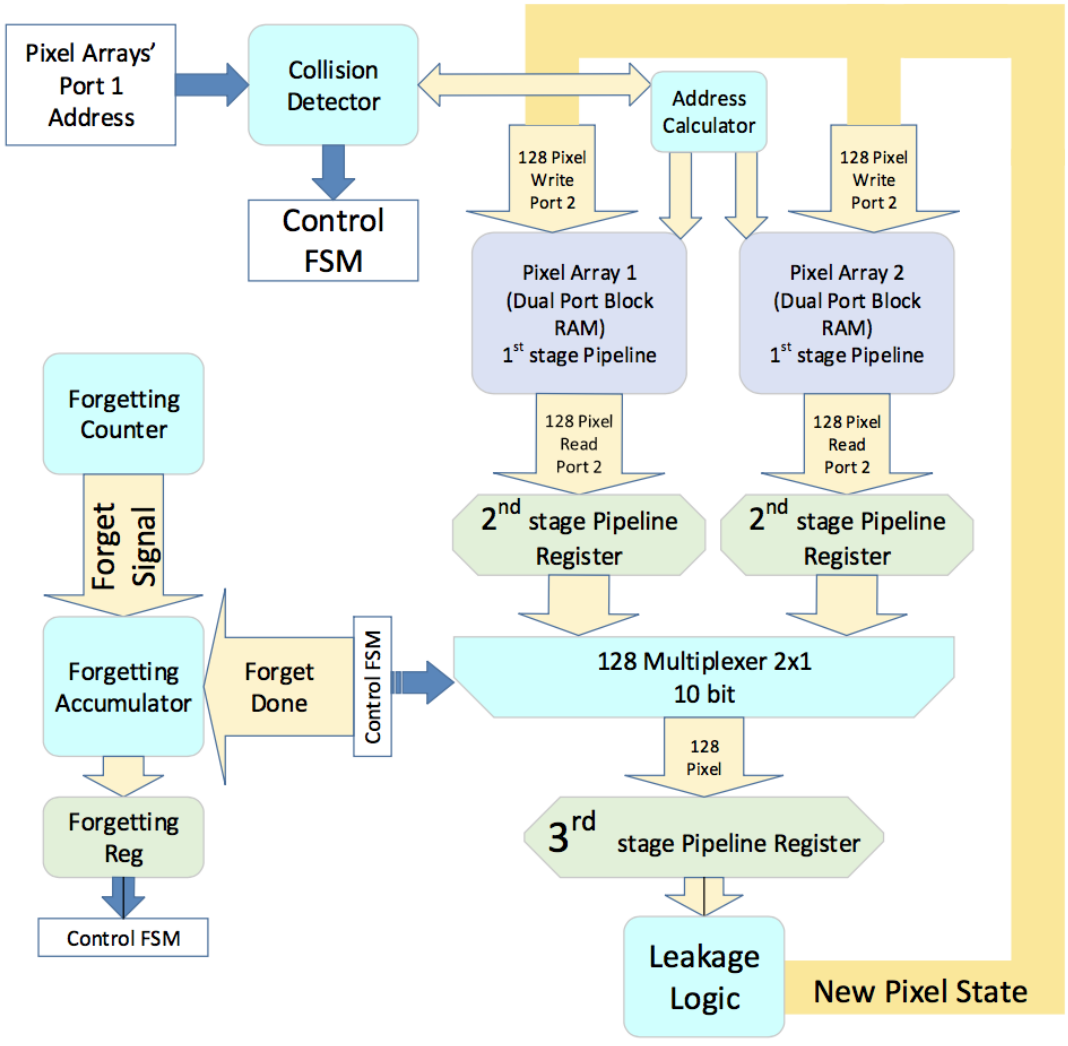


Figure 4-8: Data Flow diagram for the forgetting logic block

convolution process is just once, preventing bursts of collisions.

A complete cycle of forgetting needs 128 + 3 clock cycles if no collision happened. For each collision occurrence, the forgetting logic waits for three clock cycles and three additional clock cycles are added to fill up the pipeline again.

There is a forgetting counter in Fig. 4-8 that generates forgetting signals based on the forgetting rate defined by the user. The "forgetting accumulator" block takes care to not lose any forgetting signal within the huge traffic of input events. Whenever a forgetting signal comes, the accumulator adds '1' to the forgetting register and whenever a "forgetting done" signal activates (that means a complete cycle of forgetting has concluded), the logic will decrease by '1' the forgetting register. The forgetting register contains the number of forgetting cycles that should be performed.

The "leakage logic" block adds or subtracts '1' to the pixel value based on the sign bit. For positive values, it will decrease the number, and for negative values, it will add '1' to the value to set them closer to the reset value. For the value equal to reset, the "leakage logic" will do nothing.

4.4.3 Output event generator block

Whenever an event is generated by the "threshold logic" block of the convolution block, another part of the core takes care of these new events. This part includes two parallel processes. The first one writes the new events into the event RAM, and the second one reads them and sends them out of the core. Event RAM is a dual-port block RAM that contains 128×128 bits of data. It means that for every pixel, there is 1 bit of data in the event RAM that indicates the corresponding pixel has generated a new event or not.

Fig. 4-9 illustrates the process of writing events in the event RAM. This part also uses the same pipeline and parallel techniques and two dual port RAMs to speed up the process. Whenever a new event vector comes, the corresponding row of the event RAM will be read. The content of the event RAM row and the new event will enter into the 128 OR gates, and the result is the updated row of event RAM. The "Address calculator" logic block for this process uses the address of the pixel arrays

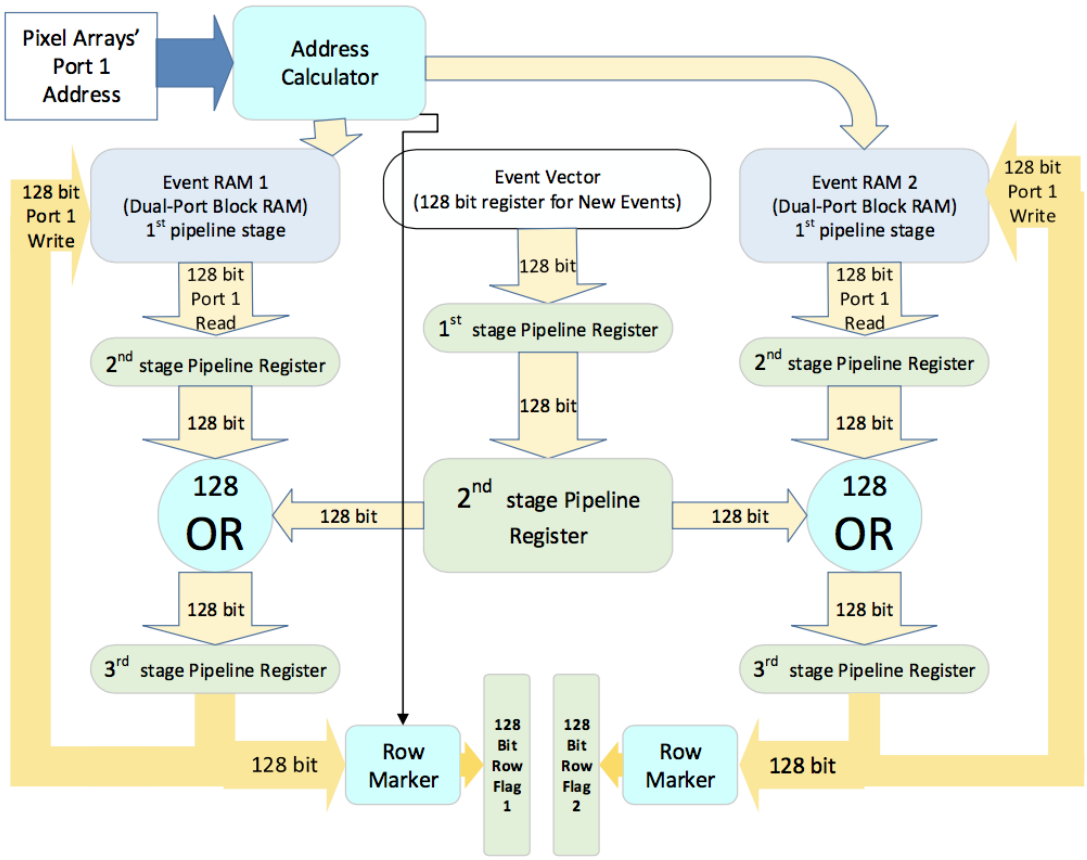


Figure 4-9: Process of writing events in the event RAM

and manipulates it to fit the new pipeline stages. The control FSM in Fig. 4-9, will control the write enable signals of the RAMs.

The "Row Marker" block is designed to help another process for output event management, which increases the speed of finding events in the RAMs. Finding an event in the whole memory by scanning each line of memory one by one is not efficient. "Row marker" calculates the OR between the 128 bits of the updated event vector and puts it in the proper column of the 128-bit row flag register. This way, each flag indicates that in the corresponding row of the event RAM, there are one or more new events waiting to be sent.

Another process to manage the generated events is the process of reading from Event RAM and send the events out of the core. Fig. 4-10 illustrates this process.

Based on the information in the Row flag registers, the row address of event RAMs will be defined through a "Row detector" logic block. Another logic block that operates in the same way is the "Event cell detector", which works on the content of the event RAM to find the events. With these 2 logic blocks, the "new event maker" logic block can find out the X and Y position of the new event to make an AER event package.

The "collision detector" is in charge of finding the write cycles with the same address in both ports of the event RAMs. In case of collision, the process of reading and sending events will always wait for the process of writing new events in the RAMs. The control logic block is responsible for asserting the write enable signals for the block RAM and taking care in collision situations. It also should handle stop signals from the output synchronous interface and propagate them back to the input synchronous interface.

Another vital role of the control FSM is sending events one by one. Whenever an event is sent, its place in the 3rd stage pipeline register should become 0 to start sending another event. The control logic does this process by changing the 3rd stage pipeline register through the multiplexer. When all of the events in the 3rd stage pipeline register have been sent, the control register will assert the event RAM write enable signal to write zero in the selected row of the event RAM and the row flag

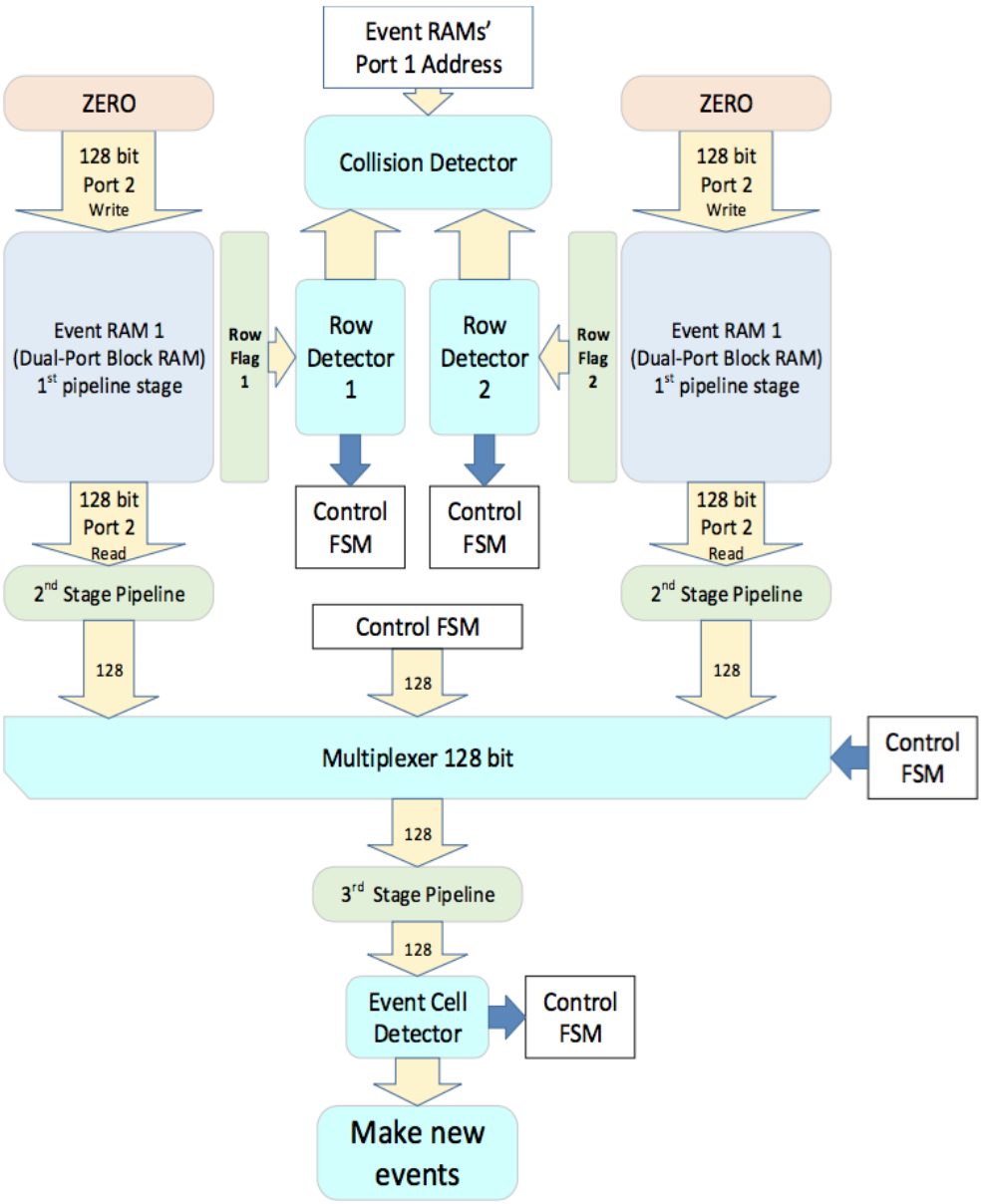


Figure 4-10: Process of reading events from the event RAM and sending events out

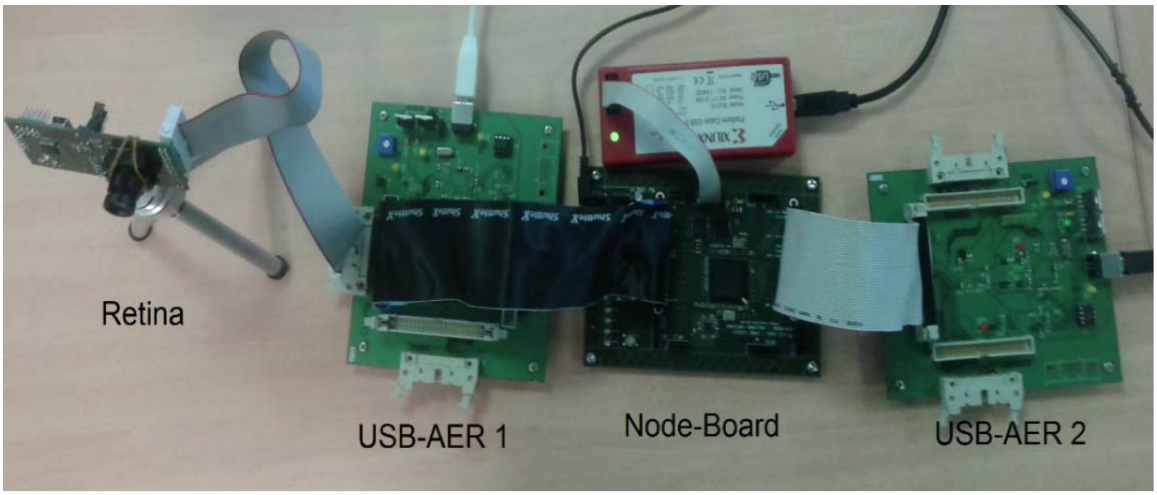


Figure 4-11: Experimental Setup

register.

4.5 Implementation Results

Fig. 4-11 shows the setup used in this work which contains a retina camera [2], a node-board [6] (containing one Spartan6 and other necessary interfaces) and 2 USB-AER boards [3] that send AER spikes (before and after processing) through USB to a computer. These boards are used to monitor DVS events, as well as convolution output events, as was shown in Fig. 4-2.

We used Xilinx XST to synthesize and implement Verilog codes. With Spartan-6 we obtained a critical path of 12ns for the pipeline stages, which allowed us to use 80MHz of the clock frequency.

Using block RAM in this project is unavoidable because of the massive amount of memory that is needed for saving neuron states. Although it is not a very expensive memory with respect to distributed RAM, it is slower, and it cannot be used in an entirely custom manner. This means it is offered in a particular size of memory. For example, in Spartan-6 the minimum size of block RAM is 9kb [58] and the width of port for dual-port RAM contains 18 bits of data. Therefore the block RAM will be 512×18 bits. The core uses these block RAMs to make a pixel array of 64×1280 bits (each pixel array contains half of pixels), so that from every 512 rows of block

FPGA Chip	Resource Utilization		
	Occupied Slices	***Occupied Block RAM***	***Critical Path***
Spartan-6 (XC6SLX45-3)	3,298 (48%)	80 x 16kb (68%)	12ns (83MHz)
Spartan-6 (XC6SLX75-3)	3,285 (28%)	80 x 16kb (46%)	12ns (83MHz)
Spartan-6 (XC6SLX150-3)	3,317 (14%)	80 x 16kb (29%)	12ns (83MHz)
Virtex-6 (XC6VLX75T-3)	4,549 (39%)	80 x 32kb (51%)	7.5ns (133MHz)
Virtex-7 (XC7VX330T-3)	4,067 (7%)	80 x 32kb (10%)	6ns (166MHz)

Figure 4-12: Resources needed in different FPGAs and maximum clock frequencies

RAM, just 64 rows have been used and there is a waste of memory happening here.

In Spartan-6 XC6SLX150T-3, the number of occupied slices is around 3.3k out of 23k, and the number of 8kb block RAMs used is 160 out of 536. Also, for comparison purposes, the core has been synthesized for Virtex-6 and Virtex-7 technologies. Fig. 4-12 shows the percentage of resources that are needed, the critical path delays and maximum frequencies in the different FPGAs.

If the kernel has L lines, the presented core needs $L+3$ clock cycles for calculating a convolution. As a comparison, Camunas [51] produced a 0.35um CMOS chip for 32×32 pixels, which for a kernel size of 23×23 the processing needed 50 clock cycles or 417ns with 120MHz clock frequency. In the same situation, the presented core requires 26 clock cycles for this kernel which in Spartan-6 needs about 312ns, in Virtex-6 about 195ns and about 156ns in Virtex-7.

Regarding other FPGA implementations of Event-Driven ConvNets, to our knowledge there are two different cases reported. Zamarreno et al. [40] used a convolution core adapted from [6], where neuron states are updated pixel by pixel, instead of row by row. This allowed for small convolution cores so that many of them could be put on one single FPGA: a total of 64 cores, each of 64×64 pixels could be put on a Virtex-6, each core together with a programmable router for configuring arbitrary ConvNets. However, as the synaptic update was pixel by pixel, it required about 3us to update one event of 11×11 convolution kernel. This is equivalent to requiring 3.17us to update a row of 128 pixels, as we are doing in this work.

Another recently reported example of event-driven ConvNets [19], announces a core update speed of 84 synaptic updates in 10ns, implemented on a Spartan6, thus achieving a performance which approaches the one reported in the present work.

4.6 Conclusion

In this chapter, we present a 128×128 pixel convolutional core for event processing that can process each kernel row in one clock cycle using a parallel and pipelined structure. In spiking ConvNet designs, using this core can speed up event processing, and it can be used to make a layer for neural networks. For convolving a kernel that contains L lines, the core needs $L + 3$ clock cycles. We implemented the core in different FPGAs. For the FPGA in our AER-Node Board (XC6SLX150-3), 12ns are needed to update the state of a 128 neuron row. The core also contains a leakage logic that works independently and does not interfere with the main process.

Chapter 5

Hybrid Neural Network, An Efficient Low-Power Digital Hardware Implementation of Event-based Artificial Neural Network

This work has been submitted to:

A. Yousefzadeh, G. Orchard, E. Stromatias, T. Serrano-Gotarredona and B. Linares-Barranco, "Hybrid Neural Network, An Efficient Low-Power Digital Hardware Implementation of Event-based Artificial Neural Network," International Symposium on Circuit and System (ISCAS), May2018

Abstract

Dynamic Vision Sensors can outperform frame-based vision sensors regarding data compression, dynamic range, temporal resolution and power efficiency. However, available mature frame-based processing methods using Artificial Neural Networks (ANNs) surpass Spiking Neural Networks (SNNs) in terms of accuracy of recognition. In this Chapter, we introduce a Hybrid Neural Network which is an intermediate solution to exploit advantages of both event-based and frame-based processing. We have implemented this network in FPGA and benchmarked its performance by using different event-based versions of MNIST dataset. HDL codes for this project are available for academic purpose upon request.

5.1 Introduction

New techniques for efficient event processing are gradually being introduced. Synaptic Kernel Inverse Method (SKIM) [59], a new learning method for synthesizing SNNs, achieved 92.87% accuracy on the N-MNIST dataset [60]. Kheradpisheh et al. [61] developed a multi-layer SNN equipped with Synaptic Time Dependent Plasticity (STDP), achieving 98.4% accuracy on the MNIST dataset [56] by converting all the MNIST frames to events through intensity to delay conversion. J.H.Lee et al. [62] developed a new method to adapt the famous error backpropagation technique for SNNs, achieving 98.66% accuracy on the N-MNIST dataset.

Even though SNNs are improving, training an efficient SNN for hardware implementation is still an open problem. This can be seen in major research initiatives involving training Artificial Neural Networks (ANNs) in frame-based domains and using trained synaptic weights for their SNN counterparts [57] [63]. Although the results of these works seem promising, they also entail serious disadvantages. Firstly, further parameter optimization is required to map from ANN to SNN, because the parameters in SNNs with Leaky Integrate and Fire (LIF) neurons (for example leak rates, threshold and refractory time) are sensitive despite the trained synaptic weights. Secondly, in these works information is coded in the event rate rather than in the exact timing of events [64] [57], so multiple events are needed to transfer information between neurons, thus increasing power consumption and delay.

One major challenge when implementing Leaky Integrate and Fire neurons is to make sure all the neurons have normal activity [63]. Another problem is premature firing before enough information (events from the previous layer) is received. Extra logic also needs to be added for inhibitory connections between neurons, to guarantee competition and eliminate high event rates. Additionally, implementing leakage in digital hardware is not a straightforward task and can be expensive (depending on accuracy).

This Chapter adopts a hybrid approach combining features of non-spiking synchronous Artificial Neural Network (ANN) and asynchronous Spiking Neural Network

(SNN). Our Hybrid Neural Network is a hardware implementation of ANN that uses event-based vision sensors as its input as described in Section 5.2. This Hybrid Neural Network has been implemented in FPGA using Hardware Description Language (HDL). We describe the experimental results of this work in Section 5.3. We have used different event-based versions of MNIST dataset to benchmark our implementation. Additionally, to compare our results with state of the art SNN hardware implementations, we have implemented on FPGA a modified version of the SNN that has been trained with the algorithm presented in [62]. A brief conclusion to this Chapter is provided in Section 7.5.

5.2 Proposed Hybrid Neural Network

The proposed Hybrid Neural Network is an ANN that uses a DVS as its input sensor. As mentioned in Chapter 1, a DVS is a power efficient vision sensor and has considerably less latency than conventional frame-based sensors. When something moves in front of a DVS, multiple pixels almost simultaneously generate events that evoke synchrony-based neural coding [65]. In this kind of coding, information is not spike rate coded or spike rank/order coded [64]. In the DVS even though neurons spike asynchronously information is coded in the simultaneous firing of a group of spikes together. To extract information efficiently, we proposed to process groups of events that are generated close together in time rather than individual events. There is some evidence that this kind of processing also takes place in the biological cortex [66]. A similar approach has been used in some efficient hardware implementations [67] [68]. A circuit called “frame-maker" was therefore designed to group the events occurring close together in time into a packet (equivalent to a frame in conventional image processing). By using this “frame-maker" after the DVS, it was possible to create an automatic adaptive frame-rate camera as our system input.

Fig. 5-1 shows a simplified block diagram of the “frame-maker" circuit. The “AER-INTERFACE" logic block converts the asynchronous communication protocol of the DVS to a synchronous protocol which is more efficient inside an FPGA [69].

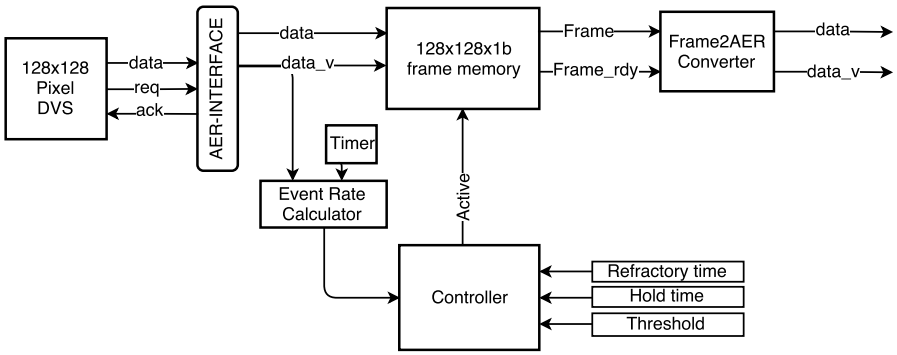


Figure 5-1: Block diagram of "frame-maker" hardware implementation. Size of frame-memory should be equal to the size of DVS or smaller (in case of subsampling DVS pixels). Frame-memory captures a binary frame. If more than an event for a specific address is received, only one event will be captured. Polarity of events can be captured in frame-memory by assigning 2 bits for each address (doubling the size of memory).

The "Frame-maker" only captures DVS events when the input event rate is higher than a given threshold[1]. That is to say a meaningful movement in front of the DVS or a saccade occurred. The frame memory block gathers all the events into a packet or binary frame after receiving an "Active" signal from the Controller (Active state). As soon as the controller deactivates this signal, the frame-memory stops registering DVS events and issues a Frame_rdy signal, indicating that the frame is ready for further processing (Non-active state). The controller is a Finite State Machine (FSM) based on a simple algorithm (see Algorithm 1).

Communications between different layers of implemented feedforward ANN is done by using AER events. In this case, each neuron puts its output value and its address into an AER packet and sends it to the next layer after receiving a specific command. This command is coded in AER format as well. We designed a "Frame2AER Converter" logic block to convert the frames into an AER packet.

To explain how our Hybrid Neural Network works, an example block diagram is shown in Fig.5-2. First, a frame-maker logic block is implemented to convert DVS events into a frame. In this network, we implemented a 28×28 pixel frame-maker

[1]To detect the event-rate, our hardware counts number of events in each millisecond, so it will take around one millisecond to detect if the event-rate passed the threshold. Then for around 10ms, all the events will be gathered in a memory. These numbers are tunable and can be adjusted based on the DVS parameters.

Algorithm 1 Frame-maker Controller algorithm. Refractory (Ref) time and Hold time are the parameters that depend on the nature of stimulus and parameters of DVS.

C1 $\leftarrow$ (Event Rate $\geq$ Threshold)
C2 $\leftarrow$ (time $\geq$ Last Non-active time + Ref time)
C3 $\leftarrow$ (Event Rate $<$ Threshold)
C4 $\leftarrow$ (time $\geq$ Last Active time + Hold time)
procedure
Non-active State:
    if *C1 & C2* **then**
        Next State $\leftarrow$ *Active State*
Active State:
    if *C3 & C4* **then**
        Next State $\leftarrow$ *Non-active State*

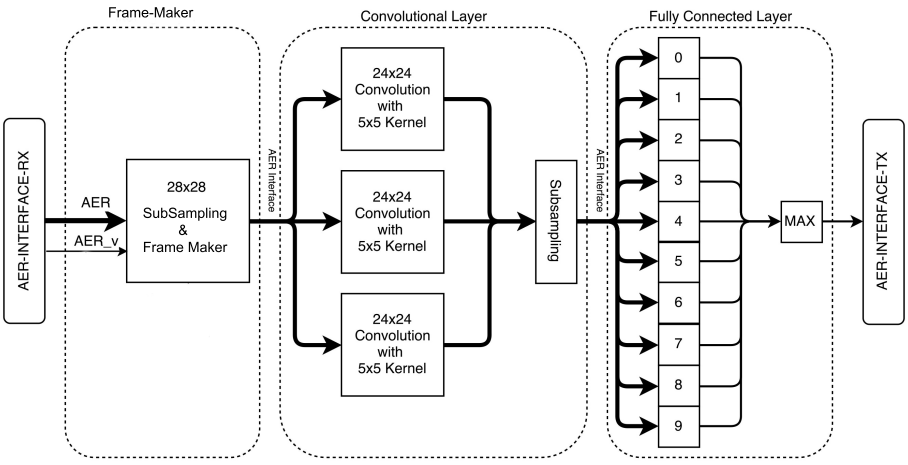


Figure 5-2: Block Diagram of FPGA implementation of HybridNet

by subsampling DVS pixels prior to storing them in the frame-memory[2]. When a frame is ready to propagate, it will be sent to the next layer. An end of frame (EOF) command event will be generated at the end of each frame.

After the frame-maker, the neural network structure should be implemented. Neurons in the Hybrid Neural Network are using the Rectified Linear unit (ReLu) as activation function [70] which can be efficiently implemented in hardware. When a layer of the neural network receives AER events from the previous layer, it will update the neurons that are connected to that specific input. After receiving an EOF command from the prior layer, each neuron with a positive membrane value sends one event to the next layer and resets to zero. Finally, an EOF event will be generated for the next layer. In this scheme neurons in a layer need to be synchronized with each other, but there is no need for synchronization between different layers. This feature (no need for global synchronization) makes it easier to implement this network in massively parallel platforms like SpiNNaker [31].

To implement the ANN we used 4-bit synaptic weights and neuron membranes. Consequently, each event contains the source address and a 4-bit parameter indicating the membrane voltage of the source neuron. Rather than using extra bits in the AER packet to encode the neurons' output, we could have used exact intensity to delay conversion (as in [61]). However, intensity to delay conversion needs to sort all neuron's membrane voltages for each layer and send them out in AER links with exact timing. Sorting thousands of numbers may decrease design performance, so we decided not to use temporal coding in this design. Casting neuron membrane values to 4-bit reduces the number of non-zero neuron outputs that need to be transmitted to the next layer.

In this work, we have implemented configurable cores for fully connected and convolutional processing. Fig.5-2 illustrates our implemented Hybrid Neural Network for handwriting digit recognition. It contains three convolutional processors with a kernel size of 5×5 in the first layer of the neural network. A 2×2 sub-sampling

[2]In addition to sub-sampling, 2×2 pixels of each border were also cropped to make frames with the exact size of the original MNIST dataset.

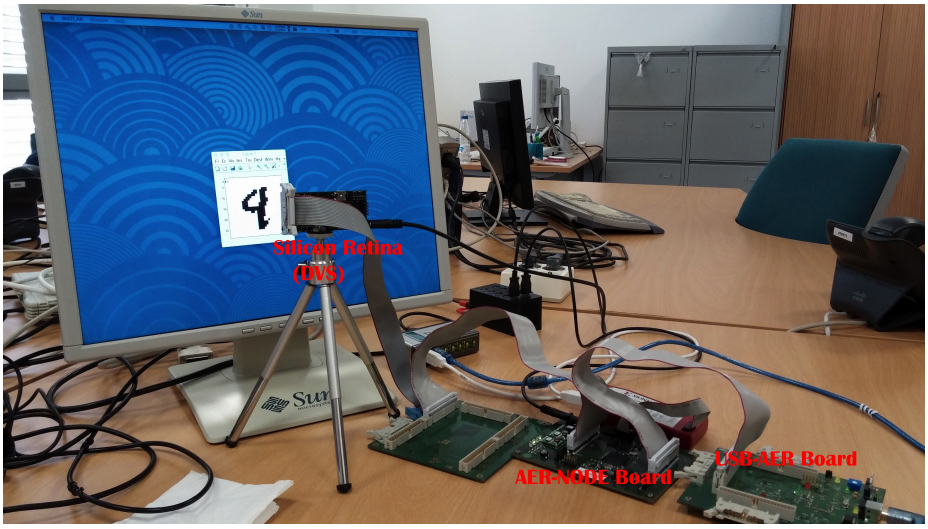


Figure 5-3: Hardware setup for real-time processing with DVS. A video of the real-time demonstration is available in [7].

layer after convolutional cores detects the most activated neuron. The last layer contains ten fully connected neurons. After processing all the input events, a MAX logic operation finds the most activated neuron and presents it as the predicted digit. To train this network, we used the TensorFlow software library [71] and downloaded the synaptic weights to the FPGA after training with 4-bit precision. In TensorFlow, inputs of ANNs are frames. To convert events of neuromorphic datasets to a frame, we have used a software model of the “Frame-maker" logic block.

5.3 Results

As mentioned earlier, we implemented a small Hybrid Neural Network in FPGA for MNIST dataset recognition. Fig.5-3 shows our hardware setup with a DVS as the input sensor, an AER-NODE board [6] with Xilinx SPARTAN-6 for implementation of the Hybrid Neural Network and a USB-AER (USBAERmini2) Board[3] for sending events back to a computer in real-time. To test the system with a pre-recorded dataset, rather than using a DVS we used an event player board [3], which played the recorded events in real-time as shown in Fig.5-4.

To report the accuracy of the network implemented in Fig. 5-2 we used three

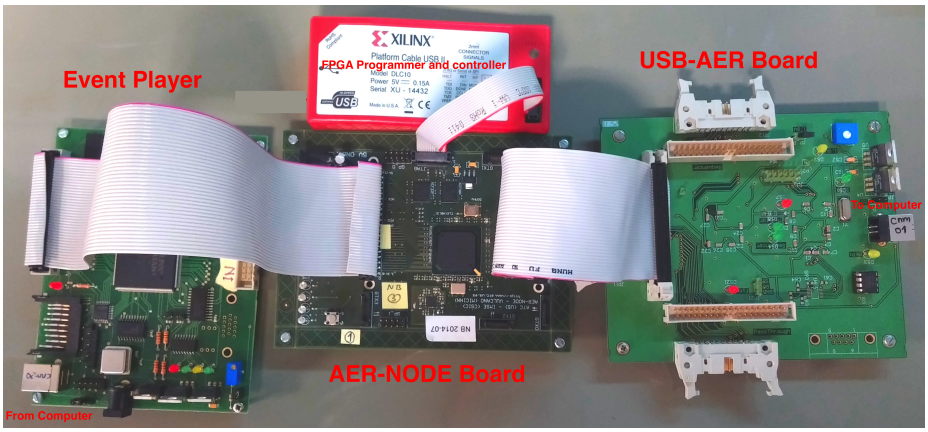


Figure 5-4: Hardware setup for processing event-based MNIST dataset by using pre-recorded events in event-player [3].

different event-based MNIST datasets. First, we used the artificial conversion of MNIST to events by generating one event per non-zero pixels from the MNIST dataset samples and no event for others. We called this dataset synthetic e-MNIST.

The second dataset that we used was the FLASH-MNIST-DVS dataset. The FLASH-MNIST-DVS dataset [72] is a full recording of the MNIST dataset obtained by flashing MNIST digits with a monitor in front of the IMSE-DVS [2]. This dataset is very similar to the artificial conversion of MNIST to events, but it also includes practical noise and statistics from a real DVS.

The last dataset is N-MNIST [60]. This dataset is captured by mounting the ATIS sensor [5] on a motorized pan-tilt unit and having the sensor move while it views the MNIST samples on an LCD monitor. N-MNIST contains three saccades for each MNIST sample. In our Hybrid Neural Network each saccade can be captured as a frame. We have trained one independent Hybrid Neural Network for each set of saccades. Therefore we implemented three Hybrid Neural Networks in parallel (one for each saccade direction). To report the accuracy, we averaged the predictions of all the three networks for each sample.

Table 5.1 shows the accuracy of the Hybrid Neural Network implemented in the FPGA. The Spartan-6 (XC6SLX150T-3) implementation of the network illustrated in Fig.5-2 works up to a clock frequency of 220MHz. The convolutional layer processes each event in '30' clock cycles (135ns). A fully connected layer needs just one clock

Table 5.1: The accuracy of FPGA implementation for two-layer Hybrid Neural Network using event-based datasets

Dataset	Train Accuracy	Test accuracy
Synthetic e-MNIST	97.91%	97.09%
FLASH-MNIST-DVS	97.99%	96.80%
N-MNIST	96.59%	96.23%

cycle for each incoming event. In the case of using one spike per non-zero pixels, each frame will be converted to '125' spikes in average. Therefore convolutional processing takes less than 17us for each frame in average. An additional 3us delay will be added by the fully connected layer and the communication system, so that processing latency for each frame is less than 20us. In a pipelined architecture every 17us a new frame can be processed by the FPGA, resulting in more than 58k MNIST frames per second. This FPGA design consumes approximately 363mW for processing 58k frames per second which is equal to less than 7uJ for each frame. Adding more layers will increase latency and power consumption but will not decrease throughput because the layers are implemented in a pipeline.

Most of the logic blocks do not need to process anything when there is no activity in front of the DVS. This implementation consumes '1270' LUTs (1.4%) and '5' Block-RAMs (1.8%) of the FPGA resources[3]

To compare an SNN implementation in FPGA with our proposed Hybrid Neural Network, we have implemented a small SNN with the algorithm presented in [62] using the MNIST dataset. In this case, we used the Poisson distribution method to convert the MNIST frames to spikes because this network does not work correctly with only one spike per non-zero pixels[4]. This SNN uses LIF neurons and contains ten convolutional populations with 5×5 kernel size in the first layer (after input) followed by a ten fully connected neurons output layer.

The original SNN presented in [62] includes complicated synaptic equations. To train the SNN in software, we used the original proposed SNN. However, in the FPGA

[3]A video demonstration of real-time handwrite digit recognition in FPGA using Hybrid Neural Network is available here [7].

[4]Around 43 events per non-zero pixel on average is generated which is recommended in the original article [62].

dynamic synapses have been replaced by static ones and exponential leakage has been implemented with bit-wise shifts [73]. We have used the same hardware setup as shown in Fig.5-4 for this experiment. Like in our previous design, for each incoming spike, '30' clock cycles were needed for the convolutional process and one clock cycle for updating the fully connected neurons. This implementation occupied '1500' LUTs and '47' Block-RAMs in our Spartan-6 FPGA, and it consumes approximately 388mW when working at full capacity (7M events per second at 220MHz). In this SNN each non-zero pixel is converted to '43' spikes in average (5k spikes for each frame in average) while we only used one spike per non-zero pixels in our proposed Hybrid Neural Network ('125' spikes for each frame in average). Consequently, this design can process less than 1.4k MNIST frames per second which results in consuming in average 300uJ per frame. The accuracy of the implemented SNN in FPGA for MNIST dataset is 97.35%[5].

5.4 Conclusion

In this work, we present a hybrid architecture for hardware implementation of ANN that uses DVS as its input. We introduce an FPGA implementation of the proposed Hybrid Neural Network that can be trained off-line by using conventional deep-learning software tools. We demonstrate that a small two-layer Hybrid Neural Network can reach 97% accuracy for MNIST dataset while consuming 7uJ per frame. Finally, we prove that this network consumes less power than state of the art SNNs in FPGA.

[5]A video demonstration of this SNN while doing handwritten digit recognition in FPGA is available in [74].

Chapter 6

Active Perception with Dynamic Vision Sensors. Minimum Saccades with Optimum Recognition

This work has been submitted to:

Amirezza Yousefsadeh, Garrick Orchard, Teresa Serrano-Gotarredona and Bernabe Linares-Barranco, "Active Perception with Dynamic Vision Sensors. Minimum Saccades with Optimum Recognition", IEEE Transaction on Biomedical Circuits and Systems

Abstract

In this Chapter, we introduce a platform for object recognition with a DVS in which the sensor is installed on a moving pan-tilt unit in closed-loop with a recognition neural network. This neural network is trained to recognize objects observed by a DVS while the pan-tilt unit is moved to emulate micro-saccades. We show that performing more saccades in different directions can result in having more information about the object and therefore more accurate object recognition is possible. However, in high performance and low latency platforms, performing additional saccades adds additional latency and power consumption. Here we show that the number of saccades can be reduced while keeping the same recognition accuracy by performing intelligent saccadic movements, in a closed action-perception smart loop. We propose an algorithm for smart saccadic movement decisions that can reduce the number of necessary saccades to half, on average, for a predefined accuracy on the N-MNIST dataset. Additionally, we show that by replacing this control algorithm with an Ar-

tificial Neural Network that learns to control the saccades, we can also reduce to half the average number of saccades needed for N-MNIST recognition.

6.1 Introduction

Dynamic Vision Sensors (DVS) are event-driven temporal contrast sensors that detect the time and pixel location of light intensity changes in the scene. For the subject to be visible to the sensor, there has to be relative motion between sensor and subject, because without motion - and under constant lighting - there would be no pixel intensity changes for the sensor to detect.

To ensure relative motion between sensor and subject, either the sensor or the subject (or both) need to be moving. In a typical sensing scenario the motion of the subject to be sensed cannot be controlled, therefore moving the sensor is a more convenient approach. This, however, poses the question of *how* to move the sensor.

For actuated frame-based vision systems, sensor motion typically involves simply pointing the sensor towards the subject using a pan-tilt unit or by mounting it on a moving platform (such as a mobile robot). However, in biological vision - by which event-driven vision sensors are loosely inspired - there is growing evidence that the motion of the vision sensors (eyes) plays a vital role in perception, and that such movement is both well controlled (albeit subconsciously) and task-dependent [75].

Examples from nature include the jumping spider, which actively moves its retina [76], the praying mantis, which executes a peering type motion for depth perception, or pigeons, which move their heads back and forth to perceive depth. Even in humans, there is growing evidence that micro-saccades during fixation play a key role in perception [65], rather than just correcting erroneous ocular drift, as was previously believed.

Intuitively, in an action-perception loop in natural animals, it is obvious that perception influences action and that action influences perception. However, most works and benchmark datasets focus on how best to perceive in order to influence action, since with pre-recorded data it is not possible to influence recordings through

actions[1]. This Chapter looks instead at how best to act to influence perception.

More specifically, this study uses the DVS, an event-driven temporal contrast sensor, to address the well-known MNIST [56] recognition task. It investigates how such a sensor should move to aid recognition in a closed action-perception loop, where the system decides what action to take next (if any) based on the sensory data it has received beforehand. We also look at whether knowledge of the action taken can be used to improve accuracy in the recognition task.

In this work have employed two different event-driven sensors. First, to compare accuracy with other published studies, a pre-recorded dataset (N-MNIST) was used [60] which was recorded using the Asynchronous Time-based Image Sensor (ATIS) [5]. Secondly, the IMSE-DVS [2] was used to demonstrate the approach in a closed loop system in real-time.

We presented static MNIST samples in front of a DVS which was mounted on a pan-tilt unit and recognized the handwritten digit by analyzing the output events of the DVS after each saccade. When we used saccades to imitate biological eye movements and object recognition in the proposed network, an interesting question arose: how is the recognition task affected by saccade direction and how many saccades are needed to recognize an object? As expected, we noticed that each saccade can contain unique information about the object related to the direction and speed of the saccadic motion.

Since performing each saccade needs a mechanical movement plus event processing, it is desirable to reduce the number of saccades while keeping the same recognition performance. After several experiments, we designed an algorithm that can suggest the direction of the next saccade based on the current information about the object. Our results show that smartly chosen saccades can reduce the average number of saccades to half in comparison to random saccades with similar recognition accuracy. Interestingly, we noticed that a neural network can perform this task and intelligently suggest saccades with almost the same performance as an analytically

[1]However, in this work we used pre-recorded data to compare our results with other works but from a different perspective. We tried to keep the performance while using less information from dataset which will be explained later in this Chapter.

developed algorithm.

Section 6.2 explains previous approaches to use the MNIST dataset with these type of sensors (or simulated sensors).

In Section 6.3 we explain our proposed approach for event-driven processing and object recognition by using a DVS, the proposed algorithm for prediction of an efficient subsequent saccadic direction for better object recognition, and how a neural network can be trained to intelligently suggest a next saccade direction.

The results of the experiments are given in Section 6.4. In this Section, we introduce our hardware-software platform for real-time object recognition with DVS saccades to demonstrate the performance of the proposed algorithm in practice. Finally, brief conclusions are provided.

6.2 Background

MNIST is arguably the most popular dataset used thus far for event-driven vision, and some different models have been applied to different event-driven variants of the original MNIST dataset. Three main event-driven versions of the MNIST dataset are used. The first is a recording of a subset of MNIST with different moving digits of different sizes presented to a DVS [77]. The second approach is to convert frames to spikes by means of intensity to delay conversion [61] or Poisson distributions [78, 62]. The most recent approach is a full conversion of the MNIST dataset at the original pixel scale, generated by moving the sensor while viewing static digits [60] (dubbed N-MNIST). This dataset is captured by mounting the ATIS sensor on a motorized pan-tilt unit and having the sensor move while it views the MNIST samples on an LCD monitor.

New techniques for efficient event processing are gradually being introduced. HOTS [79] is a new hierarchical machine learning technique that extracts visual features from events. HFirst [19] is a hierarchical Spiking Neural Network (SNN) for object recognition which uses a simple feedforward learning mechanism. This network has been implemented in low power parallel platforms such as FPGAs and

SpiNNaker [80] and achieved 71.15% accuracy on the N-MNIST dataset [60] without optimization.

Synaptic Kernel Inverse Method (SKIM) [59], a new learning method for synthesizing SNNs, achieved 92.87% accuracy on the N-MNIST dataset. Spike Time Dependent Plasticity (STDP) is an unsupervised bio-inspired learning method for SNNs. Kheradpishe et al. [61] developed a multi-layer SNN equipped with STDP, achieving 98.4% accuracy on the MNIST dataset by converting all the MNIST frames to events through intensity to delay conversion. J.H. Lee et al. [62] developed a new method to adapt the famous error backpropagation technique for SNNs, achieving 98.66% accuracy on the N-MNIST dataset.

N-MNIST contains three saccade recordings for each MNIST sample. Based on our knowledge, no research has been done to improve the performance of recognition by considering each of these saccades as an individual source of information which is coupled to the direction of the saccade. We have used the N-MNIST dataset to benchmark performance of our proposed method but in an entirely different perspective. While it makes sense to use all the three saccades from each sample to improve recognition accuracy, in real-time robotic applications each additional saccade comes with a cost in power consumption and recognition latency. Therefore when we used N-MNIST, we considered the cost of each saccade along with the recognition accuracy and tried to use fewer saccades to recognize the handwritten digits.

6.3 Event-Driven Recognition with Saccades

In this Section, first, we explain the methods that we used for processing saccadic events in a neural network. This neural network is an efficient feedforward network that receives DVS events and processes them to recognize handwritten digits. The output of this network is a prediction vector that contains ten values (one for each digit). Later on, in Section 6.3.2, we integrate this network into a closed-loop system along with a block to control the direction of saccadic motion. This block, which we call the “Next Saccade Prediction" (NSP) block tries to suggest an optimal direction

for the subsequent saccade based on the current output of the feedforward handwritten digit recognition network. The NSP block executes an analytical algorithm to suggest the next saccade direction. In Section 6.3.3 we show how the NSP block can be replaced by a neural network. In this case, both the feedforward and feedback processing will be done by using neural networks.

6.3.1 Feedforward Symbol Recognition Neural Network

As mentioned earlier, a DVS is power efficient and has considerably less latency than conventional frame-based sensors. However, it is generally harder to extract information from the DVS events by using conventional image processing methods. To extract information efficiently, we propose processing groups of events that are generated close together in time rather than processing individual events. This is not a new idea and has been used in efficient hardware implementations [67, 68]. There is evidence that this kind of processing also takes place in biological cortex [66]. A block which is called “frame-maker" was therefore designed to group events occurring close together in time into a packet (equivalent to a frame in conventional image processing). By using such a “frame-maker" after DVS sensing, it was possible to create an automatic adaptive frame-rate camera for our system input.

To build a frame from each saccade, one approach is to put a fixed number of events in a frame [68]. This method may result in multiple frames for a saccade. Also, each stimulus may need a variable number of events to construct a precise frame. For example, digit ‘8’ is bulkier than digit ‘1’ and will require more events to have a clear frame.

In the N-MNIST dataset [60], each saccade takes about 100ms. A saccadic movement has the highest velocity in the middle of the saccade. Therefore output event rate is maximum around 50ms after the start of a saccade. Fig. 6-1 shows the average event rate of one saccade in the N-MNIST dataset. Our experiments show that collecting events during a short time when the event rate is high will result in a sharp frame. As it is illustrated in Fig. 6-1, a frame for each N-MNIST saccade can be created by collecting all the events which are generated in a time span of 10ms

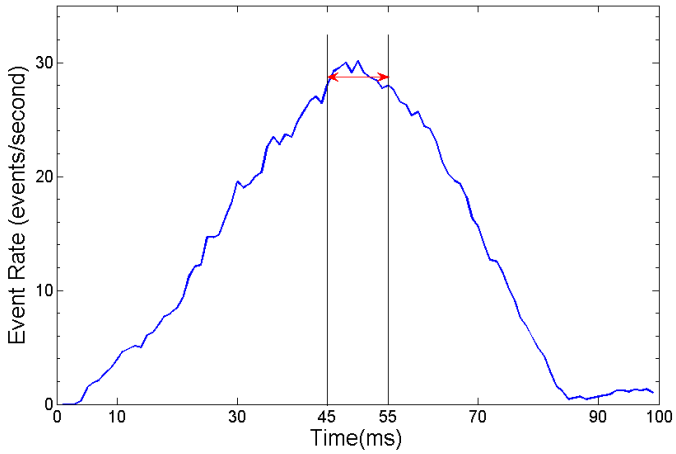


Figure 6-1: Average event rate of a saccade in N-MNIST dataset per each millisecond. A frame is constructed by integrating the events in the time span of +/-5ms (between 45ms to 55ms) around the peak average event rate at 50ms.

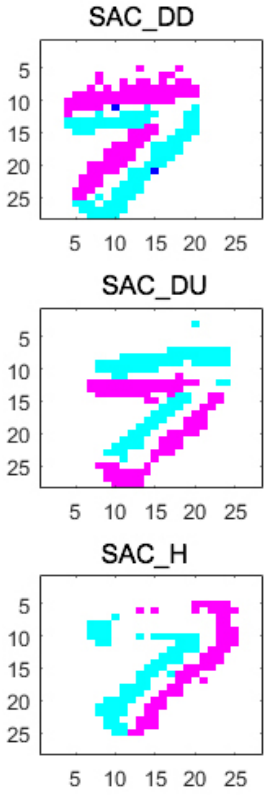


Figure 6-2: Three saccades captured from sample '80' of test set in N-MNIST dataset. The colors show the polarity of the events. Blue is for negative events and purple is for positive events. Dark blue indicates places where both positive and negative events occurred

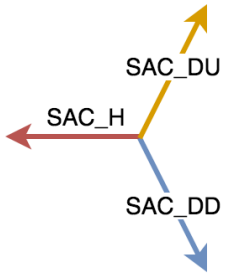


Figure 6-3: Direction of saccades in the N-MNIST dataset, SAC_DD (Diagonal Down Saccade), SAC_DU (Diagonal Up Saccade) and SAC_H (Horizontal Saccade).

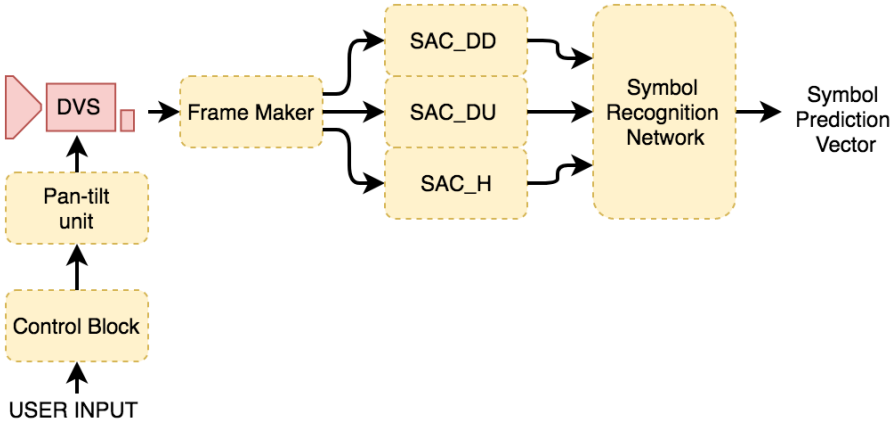


Figure 6-4: Block diagram of proposed feedforward system for symbol recognition with saccades, using a DVS. The DVS is connected to a moving pan-tilt unit. The "frame-maker" block assembles a frame after each saccade and then stores that frame in its corresponding memory (SAC_DD, SAC_DU or SAC_H). Here we limit the direction of movements to three to remain compatible with the N-MNIST dataset. The Control Block is responsible for controlling the direction and speed of the pan-tilt unit movements based on the user input.

around the center of a saccade. The events outside this time will not be processed. Fig. 6-2 shows three frames that are generated by three saccades of a sample in the N-MNIST dataset. Directions of these three saccades are shown in Fig. 6-3.

Fig. 6-4 shows the block diagram of the system that has been used to perform symbol recognition with a DVS through saccadic movements. The output of the "frame-maker" block is a 28×28 pixel binary frame[2]. After the "frame-maker" block,

[2]There is only one bit for each pixel in this frame, so multiple events with the same address will not carry additional information. To save power, it is recommended to adjust the DVS parameters (like threshold, refractory period, ...) and pan-tilt unit parameters (like the velocity and range of movement) in a way that each pixel generates maximum one event for each frame.

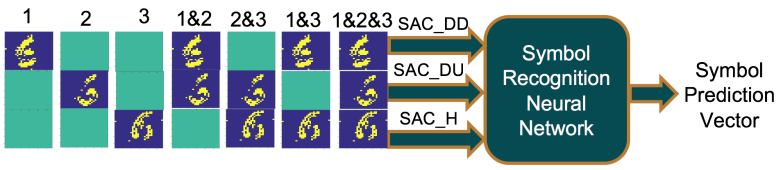


Figure 6-5: "Symbol Recognition Neural Network" (SRNN) with one, two and three saccades for a sample of N-MNIST dataset

a conventional Artificial Neural Network (ANN) is implemented to process the frames and recognize the handwritten digits. This ANN is called "Symbol Recognition Neural Network" (SRNN) and receives a hyper-frame as its input. The hyper-frame is a frame with 28×84 ($28 \times 28 \times 3$) pixels and can contain all the frames made by the three saccades of each sample. This network should be able to work with one, two or three saccades of each input sample, as illustrated in Fig. 6-5. Therefore, during the training phase, SRNN was trained to accommodate all possibilities. This means that the SRNN can use one, two or all three saccades of a sample to predict the presented digit[3]. We used a blank input in the position of the non-available frames to construct the 28×84 pixels of a hyper-frame. The SRNN can have an arbitrary number of layers with various architectures (convolutional, fully connected, etc.). In this work, we tried a few small but accurate enough neural networks to perform our experiments.

The SRNN outputs a prediction probability $\hat{y}_i$ for each class i. The recognized digit is the one with the highest prediction value. The sum of all the values in a prediction vector is normalized to one. Therefore, each value can be interpreted as a probability. The quality of the prediction vector can be measured by calculating the following "prediction loss function"

$$\mathcal{L} = \sqrt{\frac{\sum_i (\hat{y}_i - y_i)^2}{2}} \tag{6.1}$$

where $\mathcal{L}$ is the loss, $\hat{y}_i$ is the prediction value for class i, y_i is the ground truth label for class i represented using one hot encoding ($y_i = 1$ if the class is i, and zero otherwise).

[3]Recognition accuracy increases (on average) when more saccades are provided.

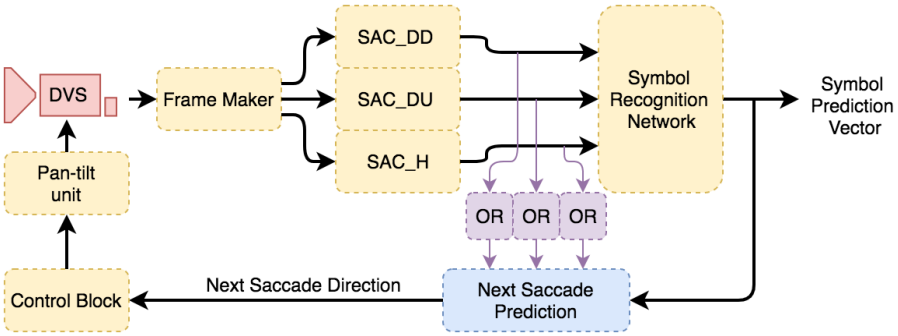


Figure 6-6: Block diagram of the closed-loop recognition system. The NSP block calculates the best direction for the next saccade, if it is needed.

6.3.2 Closed Loop Recognition with Analytical Algorithm

In biological vision, it is the movement of the retina (saccades and micro-saccades) that enables one to see clearly [65]. In our study, we used a standard pan-tilt platform to move the DVS while the object in front of it was fixed. The information obtained from saccades is determined by the movement parameters. A movement can be described in terms of its velocity, distance, and direction. The movement velocity can affect the rate at which events are generated. For a clear saccade to be captured, the movement has to be sufficiently fast over a short distance. The recognition task can also be affected by the direction of the movement. For example, horizontal saccades intensify vertical edges but suppress horizontal ones. This also influences the relative positions of positive and negative events, leading to different perceptions of the same object (see Fig. 6-2).

Intuitively, two strong enough saccades of a DVS with different directions should be sufficient to retrieve all the information from a two-dimensional picture. For objects without any prominent edges parallel to the direction of the saccade, just one saccade could be enough for recognition. An extra saccade increases power consumption and delay in recognition, but it may also provide additional information about the object. In real applications, a robot can choose to perform an extra saccade or not, depending on its current knowledge of the object. The same decision can be made regarding the direction of the saccade.

As shown later in Section 6.4, we noticed that more than 94% of the test samples in

the N-MNIST dataset could be recognized correctly with only one saccade. Therefore, performing an extra saccade for those samples represented a waste of time and power. In this Section, we look at how we can predict the need for an additional saccade and the best direction for it. We added another block to Fig. 6-4, called NSP. This block closes the loop of our system, as shown in Fig. 6-6.

Fig. 6-6 shows the inputs and output of the NSP block. One of the inputs is the $\hat{y}$ which is generated by processing one or more saccades. The other three inputs to the NSP block indicate which saccades have contributed to recognition. This information is important to avoid suggesting an already performed saccade again. An 'OR' logic can determine which memory block contains non-zero pixels. Memory blocks which correspond to the saccades that have not been executed contain all zero values.

The output of the NSP block determines the next saccade direction. The next saccade direction can be one of these four possibilities (see Fig. 6-6):

1. No extra saccade
2. SAC_DD
3. SAC_DU
4. SAC_H

The NSP block is implemented in a closed loop with the SRNN to perform the following tasks in the order shown:

1. Receive events from the first saccade and make the first guess about the object
2. Predict the best direction for the next saccade, if necessary
3. Command the pan-tilt unit to perform the next saccade
4. Combine information from all the saccades performed so far to improve recognition accuracy
5. Continue from step 2)

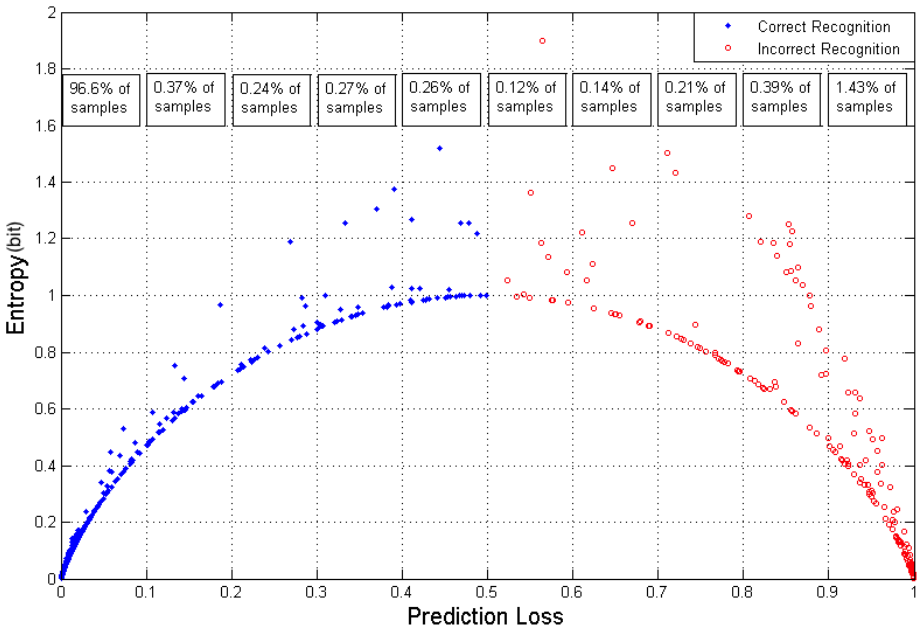


Figure 6-7: Relationship between entropy and prediction loss for the 10,000 test samples of the N-MNIST dataset. A two layer SRNN has been used. A sample is classified as correctly recognized when the position of the maximum value in its prediction vector correctly shows the class of the presented digit. For each 0.1 interval in the x-axis, we indicate the percent of test samples within this interval (see boxes on the top part of the figure).

Our experimental results showed that if the SRNN is not sure about the recognition results, the NSP block should request extra saccades. To quantify the amount of uncertainty in a prediction vector, we used the definition of entropy in information theory

$$\mathcal{H} = -\sum_{i} \hat{y}_i log_2(\hat{y}_i) \tag{6.2}$$

Fig. 6-7 illustrates the relationship between entropy and prediction loss (see eq. (6.1)) for the test samples in the N-MNIST dataset for a specific SRNN. It shows that having a small entropy cannot guarantee a correct recognition. Sometimes, it is possible that the SRNN can be very confident but the answer is wrong. Our experiments show that in these cases, performing extra saccades cannot help to find the correct answer and the only solution is to improve the SRNN[4]. High entropy in

[4]In this work we do not intend to improve the capability of the SRNN with novel techniques,

a prediction vector means less certainty about the results and this is the time when performing extra saccades might be helpful[5].

The NSP block can decide to ask for an extra saccade when the entropy is higher than a predefined threshold θ_H. This θ_H is a user-defined parameter and depends on the cost of an additional saccade. Obviously, choosing a smaller θ_H will result in performing more saccades on average and increasing the average recognition accuracy. In Section 6.4 the relationship between θ_H, the average number of saccades and the recognition accuracy for our experiments will be explained in detail.

If the NSP block asks for an extra saccade, another mechanism will also be needed to define the best choice among different saccade directions. For this purpose, we decided to extract some statistics from the training set of the N-MNIST dataset.

Before performing any saccade, our system does not have any information about the presented object. In this case, the NSP block chooses the saccade that shows the best average performance among all the three saccades during training. For each of the possible next saccade directions, we have defined a vector which is called "Confidence Coefficient Vector" (CCV).

To explain how to calculate the CCVs, suppose that the first saccade is SAC1. The possible next saccades are SAC2 and SAC3. The CCVs for SAC2 and SAC3 can be calculated by the following steps:

1. P1 = Prediction vectors for all the training samples after presenting SAC1 [it is a $10 \times 60,000$ matrix]

2. L12 = Prediction losses for all the training samples after presenting SAC1 and SAC2 [it is a $1 \times 60,000$ vector]

3. L13 = Prediction losses for all the training samples after presenting SAC1 and SAC3 [it is a $1 \times 60,000$ vector]

rather we would like to use the available network as efficiently as possible.

[5]For example, when the prediction vector is [1,0,0,0,0,0,0,0,0,0] entropy is zero, while it is maximum (3.32) when the prediction vector is [0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1]. In the first case, the network is confident that the presented sample is digit '0' while in the second case, the network is not sure about any of the classes.

4. P12_Best = P1 where L12 is smaller than L13 [it is a $10 \times x$ matrix where x is the number of training samples which have smaller L12]

5. P13_Best = P1 where L13 is smaller than L12 [it is a $10 \times y$ matrix where y is the number of training samples which have smaller L13] [6]

6. CCV2 = mean(P12_Best) [it is a 10×1 vector]

7. CCV3 = mean(P13_Best) [it is a 10×1 vector]

In other words, the CCV is the average of the prediction vectors for the training set samples with the minimum prediction loss for a pre-defined next saccade. CCVs are calculated once, after training the SRNN. This method can easily be extended for more than three saccades.

The NSP block chooses the next saccade which has the most similar CCV to the current prediction vector. In this way, we hope that the upcoming saccade will be in a way that is best for the existing guess of the SRNN. The similarity of two vectors can be calculated by measuring their Euclidean distance. This method, even though it is not a very precise method for choosing the next saccade direction, is nevertheless quite simple.

The algorithm for the NSP block can be summarized as follows:

1. The first saccade is always the one that shows the best average results for the training samples.

2. Once the entropy (see eq. 6.2) of the prediction vector of the first saccade has been calculated, it is compared with θ_H to see whether an extra saccade is needed or not.

3. If an extra saccade is needed, the best saccade can be found using the CCVs

4. Calculate the entropy again and compare it with θ_H to see whether an extra saccade is needed or not.

5. Continue from step 3)

[6] $x + y = 60,000$

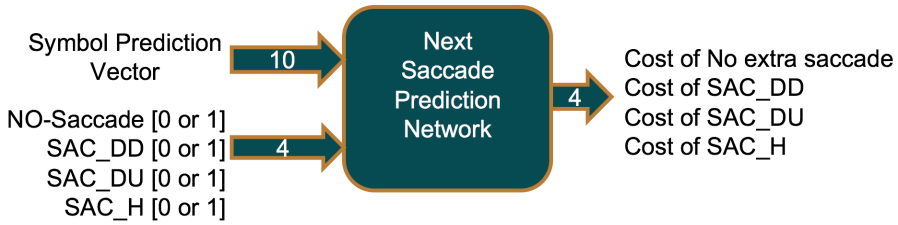


Figure 6-8: Inputs and outputs of "Next Saccade Prediction Network" (NSPN)

6.3.3 Closed-Loop Recognition with a Neural Network

Next, we investigated the use of an ANN to predict which saccade should be performed next. In this case, both feedforward and feedback paths will be equipped with neural networks which reduce the complexity of the system. Therefore we replaced the proposed algorithm by a neural network inside the NSP block in Fig. 6-6.

Fig. 6-8 shows the inputs and outputs of the "Next Saccade Prediction Network" (NSPN). Inputs and outputs of this network are highly compatible with the previously presented NSP block. The "Symbol prediction vector" is the output of the SRNN. The other four inputs indicate the currently executed and available saccades. Previously, in the NSP block we only used three inputs for providing this information, which was enough. However, during training, we noticed that the neural network can learn better if the inputs are normalized. We wanted to train the network to predict the best saccade direction for the initial movement as well, so we decided to add an active input for this case which is called "NO-Saccade". The "NO-Saccade" input is equal to the 'NOR' of the other three inputs (SAC_DD, SAC_DU, SAC_H).

The NSPN outputs are four values which represent a "cost" for each action. In the current implementation, the pan-tilt unit will move in the direction with minimum predicted cost (Hard-Threshold)[7]. For example, when the value of output "Cost of No extra saccade" is the minimum, it means the NSPN is suggesting not to do any further saccade, because the current information about the object might be enough.

To train this network, we calculated a "cost" for each action for all the possible

[7]Another suggestion is not to restrict the DVS to move in the direction of one of these three saccades, but to let it move in a mixture of directions based on the cost of each saccade (Soft-Threshold).

combinations of inputs and used it for supervised learning. To calculate this “cost", first, we determined the power and latency cost of an additional saccade. This value, which we call “saccade cost" (or “mechanical cost") $Cost_{saccade}$, is equivalent to the θ_H in the previously introduced NSP block. When this value is high, the NSPN is more likely to suggest no extra saccade for the next action. Here, we used the same $Cost_{saccade}$ for all three saccades. Another critical parameter to determine the cost of each action is how much this action will help to reduce the “prediction loss" (see eq. (6.1)).

We define the “cost" of each action as the sum of the “prediction loss" value (which is calculated after performing the action) and the $Cost_{saccade}$

$$Cost = Cost_{saccade} + \mathcal{L} \tag{6.3}$$

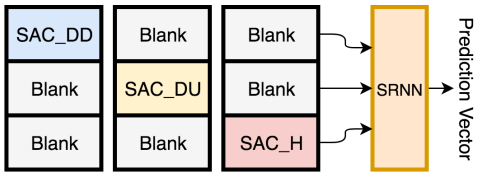
For the “No extra saccade", $Cost_{saccade}$ value is zero. Otherwise, it is a predefined constant value. From the four possible next actions, the system picks the one whose output predicts the minimum cost. Our goal is that the system learns to reduce the total number of saccades required to achieve the same recognition performance.

6.4 Experimental Setups and Implementation Results

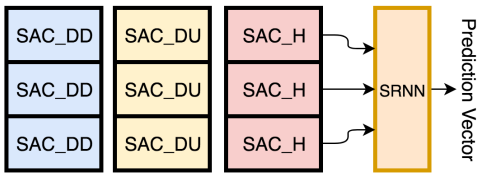
Section 6.4.1 describes the results of our experiments for saccade-based recognition with the DVS moving in a predefined direction. Later on, we describe the results for closed loop recognition, when DVS movement was controlled by the NSP block (Section 6.4.2) and the NSPN (Section 6.4.3).

6.4.1 Feedforward Recognition with Saccades

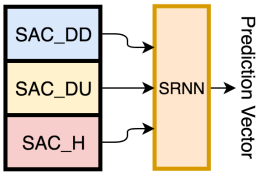
This Section reports only the test results for the open loop system in Fig. 6-4 when the DVS moves in a predefined direction (i.e., no saccade prediction). For these experiments we used the pre-recorded N-MNIST dataset [60].



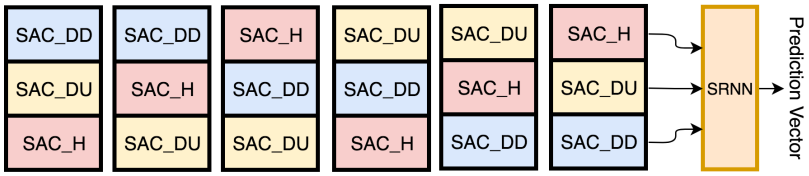
A. Experiment 1 (with direction information)



B. Experiment 2 (without direction information)



C. Experiment 3 (with direction information)



D. Experiment 4 (without direction information)

Figure 6-9: Different experimental configurations for testing whether feeding direction information can be helpful for learning.

Our goal here is to study under what condition it is relevant to provide specific information about direction of provided saccades to improve recognition. To do so, we designed the experiments shown in Fig. 6-9. In experiments '1' and '3', input hyper-frames are built by allocating saccades in the same positions, while for experiments '2' and '4', saccade positions are intentionally shuffled. Let us call these saccade positions "channels".

To see the effect of network size, three different network sizes for the SRNN were implemented: a 3-Layer network (3C5x5-128FC-10FC), a 2-Layer network (3C5x5-10FC) and a 1-layer network (10FC) [8].

The spikes' polarity bits reveal information about the movement direction. To find out the effect of using spikes' polarity, we have done all the experiments with and without using spikes' polarity. Each experiment was carried out once using only the positive polarity events and once using both polarity events. When only the positive polarity was used, input size was $28 \times 28 \times 3$, while when using both polarities input size was $28 \times 28 \times 3 \times 2$. In the second case, the network size was therefore larger.

By comparing experiments '1' and '2' in Fig. 6-9 we wanted to find out if it helps or not to feed all saccades through the same channel. To have a fair comparison, we always used the same input size, but the hyper-frame arrangements are different. In this case, accuracy is calculated by averaging the prediction vectors of all three saccades.

By comparing experiments '3' and '4' in Fig. 6-9 we wanted to find out the effect of feeding explicit information about direction when all three saccades are available. In experiment '3' a hyper-frame contains only one arrangement of saccades (SAC_DD/SAC_DU/SAC_H) while in experiment '4' all six possible shufflings are provided.

We also tested the speed of learning. We report the accuracy of the networks after 3 training epochs and also after 50. It should be noted that the number of training samples in each epoch in the different experiments was not equal. While there are

[8] FC indicates a Fully Connected layer while C indicates a Convolutional layer. For example, 3C5x5 means a convolutional layer with three feature maps and kernel size of 5×5 and 10FC means a fully connected layer of 10 neurons.

Table 6.1: Accuracy of experiments in Fig. 6-9 for different network sizes, using only positive events, after '3' epochs.

	1-Layer	2-Layer	3-Layer
Experiment 1	91.9%	95.5%	97.7%
Experiment 2	70.5%	83.9%	89.2%
Experiment 3	95.0%	96.9%	97.2%
Experiment 4	82.1%	95.0%	97.3%

Table 6.2: Accuracy of experiments in Fig. 6-9 for different network sizes, using only positive events, after '50' epochs.

	1-Layer	2-Layer	3-Layer
Experiment 1	92.6%	97.3%	98.3%
Experiment 2	85.2%	89.4%	97.9%
Experiment 3	95.4%	97.2%	98.6%
Experiment 4	89.4%	96.4%	98.1%

60,000 training samples in the N-MNIST dataset, in experiments '1' and '2' we had 180,000 (60,000×3), in experiment '3' we had 60,000 and in experiment '4' we had 360,000 (60,000×6) hyper-frames in each epoch.

Tables 6.1, 6.2, 6.3 and 6.4 show the results obtained in all the Fig. 6-9 experiments. Based on the results we concluded the following:

1) Feeding explicit direction information accelerates learning. This can be seen by comparing the accuracy of the networks after 3 and 50 epochs. For experiments '1' and '2' which had the same number of hyper-frames in each epoch, Table 6.5 shows the difference between accuracies after 3 and 50 epochs. It can be seen that by using explicit direction information (experiment '1'), the training process converges faster.

2) When network size is smaller, feeding explicit direction information (experi-

Table 6.3: Accuracy of experiments in Fig. 6-9 for different network sizes, using both positive and negative events, after '3' epochs.

	1-Layer	2-Layer	3-Layer
Experiment 1	94.8%	97.4%	97.2%
Experiment 2	74.7%	84.2%	95.1%
Experiment 3	96.3%	95.8%	97.8%
Experiment 4	80.1%	96.6%	97.6%

Table 6.4: Accuracy of experiments in Fig. 6-9 for different network sizes, using both positive and negative events, after '50' epochs.

	1-Layer	2-Layer	3-Layer
Experiment 1	95.4%	97.8%	98.6%
Experiment 2	89.0%	95.2%	98.6%
Experiment 3	96.5%	98.0%	98.8%
Experiment 4	89.8%	97.0%	98.1%

Table 6.5: Accuracy difference between '50' and '3' training epochs, for experiments '1' and '2' in Fig. 6-9, when using only positive events.

	1-Layer	2-Layer	3-Layer
Experiment 1	0.7%	1.8%	0.6%
Experiment 2	14.7%	5.5%	8.7%

ments '1' and '3') improves recognition accuracy. This indicates that larger networks with more learning capacity can extract saccade directions from each frame themselves without the need of explicit information[9]. Table 6.6 shows the difference between accuracies of experiments '1' and '2' and between accuracies of experiments '3' and '4'. It shows that when network size is larger, the difference between accuracies drops.

3) Using polarity of spikes improves the recognition accuracy. This can be because of multiple reasons. First, the network size is larger in this case (input size is twice, which results in more synaptic connections for the input layer). Second, there is additional information about the object in the negative polarity spikes. This means, even though negative polarity spikes contain very similar information than the positive

Table 6.6: Difference of accuracies between experiments '1' and '2' and between experiments '3' and '4' in Fig. 6-9 after 50 epochs when using both polarities.

	1-Layer	2-Layer	3-Layer
Experiment (1)-(2)	6.4%	2.6%	0.0%
Experiment (3)-(4)	6.7%	1.0%	0.70%

[9]To investigate more about this fact, we have done another experiment. In this test, we trained three neurons (each corresponds to one saccade direction) to predict the direction of a saccade. The input of each neuron was a 28×28 pixel frame. We found out that this network could predict the saccade direction (which the input frame is made of) with more than 98% accuracy without using spikes polarity.

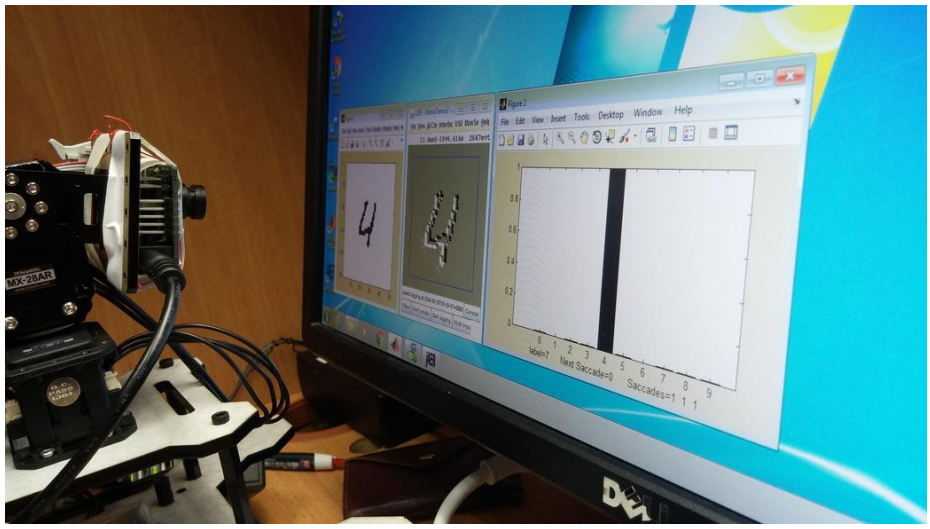


Figure 6-10: Hardware setup for moving the DVS with a pan-tilt unit

polarity ones, they are not a perfect copy of each other. The third reason is that using both polarities at the same time contains information about the movement direction.

4) In experiments '1' and '2', the final prediction vector for each sample is the average of the prediction vectors for all three saccades. In experiments '3' and '4', the neural network receives all the saccades and mixes them. Experimental results show that, in general, averaging the prediction vectors for all three saccades is not always the best strategy and a neural network itself may find a more optimized way to mix the prediction vectors.

6.4.2 Using Closed-Loop Next Saccade Prediction Algorithmic Block

Fig. 6-10 shows the hardware setup with the DVS mounted on a mechanical pan-tilt unit so that it can be moved in a desired direction. We have used this setup for a real-time demo [81], however, for reporting the next results, we have used the N-MNIST dataset, so that interested readers can reproduce them.

For the closed loop recognition experiments, we selected a 2-layer SRNN (5C5x5-10FC) with the configuration of experiment '3' in Fig. 6-9 and we used only positive polarity events. Table 6.7 shows the average accuracies of the SRNN for each saccade

Table 6.7: Accuracy of two-layer(5C5x5-10FC) SRNN for the different combinations of input saccades with N-MNIST dataset (only positive polarity events are used)

Input saccades	Training Accuracy	Testing Accuracy
SAC_DD	95.8%	94.4%
SAC_DU	95.5%	93.6%
SAC_H	96.6%	94.5%
SAC_DD and SAC_DU	98.4%	96.9%
SAC_DD and SAC_H	98.7%	97.1%
SAC_DU and SAC_H	98.7%	97.0%
All Saccades	99.3%	97.7%

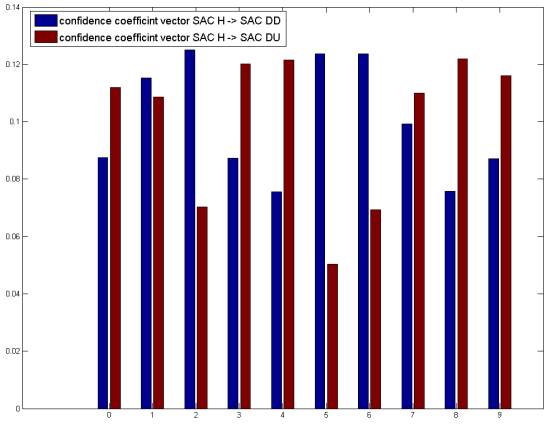


Figure 6-11: Confidence coefficient vectors (CCVs) of SAC_H→SAC_DD and SAC_H→SAC_DU

combination of the N-MNIST dataset. As can be seen, adding an extra saccade always increases accuracy.

From Table 6.7, it can be seen that SAC_H has the best average accuracy among all three saccades of the training samples. Therefore, the NSP block always choses SAC_H as the initial saccade. The next action can be "No extra saccade", "SAC_DD" or "SAC_DU".

If the entropy of the prediction vector is higher than θ_H, the NSP block should choose the "SAC_DD" or "SAC_DU" as the second saccade. In Section 6.3.2 we explained the method to extract CCVs. The confidence vector for SAC_DD after

SAC_H, for example, can be calculated by averaging the prediction vectors[10] of those training samples that showed the best results when SAC_DD was performed after SAC_H. Fig. 6-11 shows the CCVs of SAC_H→SAC_DD and SAC_H→SAC_DU. These vectors show which saccades are better for which classes. In Fig. 6-11, for example, it can be seen that for digit '5' SAC_H→SAC_DD has more confidence than SAC_H→SAC_DU.

Fig. 6-12 shows the results of the analytical approach to saccade prediction using different θ_H. As expected (Fig. 6-12 A and B) the average number of saccades and the accuracy decrease by increasing θ_H. Fig. 6-12(C) shows the relationship between accuracy and the average number of saccades for the different θ_H. For example, if θ_H is set at '0.09' the average number of saccades will be '1.54', while accuracies for training and testing samples will be 99.24% and 97.57%, respectively. By looking at Table 6.7, we notice that this is almost equal to the result of the open loop recognition with three saccades per sample. This means that by using the NSP block, it is possible to reach the highest possible accuracy of the SRNN while only performing half of the number of saccades on average.

6.4.3 Using Closed-Loop Next Saccade Prediction Neural Network

This Section shows the results obtained with the closed loop network configuration with a neural network in the feedback path. In this experiment, we used the same SRNN which was used in Section 6.4.2 and we only replaced the NSP block by a neural network (which we call "Next Saccade Prediction Network" NSPN). This network is shown in Fig. 6-8 and was trained using the training samples of the N-MNIST dataset[11]. We used a small but fully connected 4-layer network (50FC-50FC-50FC-4FC) for the NSPN.

The NSPN was trained with different $Cost_{saccade}$. Fig. 6-13 shows the network

[10]These prediction vectors are calculated only after presenting SAC_H.

[11]The NSPN is using the output of the SRNN. Therefore, the NSPN will be trained after the SRNN.

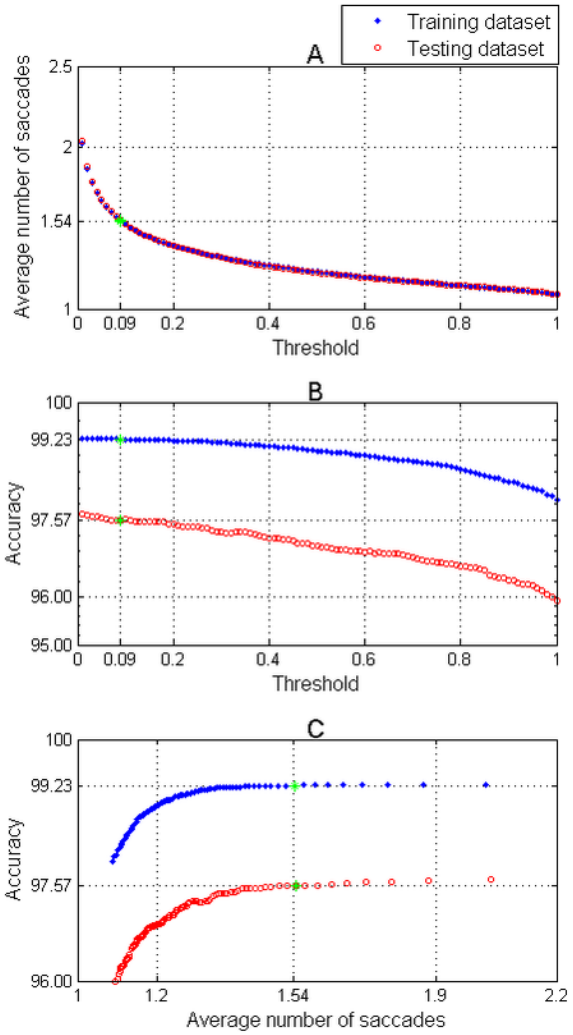


Figure 6-12: Results of analytical approach to saccade prediction with different entropy thresholds (θ_H). Green marks show the accuracy and average number of saccades when θ_H is '0.09'. For this θ_H accuracy is close to the highest accuracy of our SRNN, while instead of using all 3 saccades per sample, only 1.54 saccades in average are used.

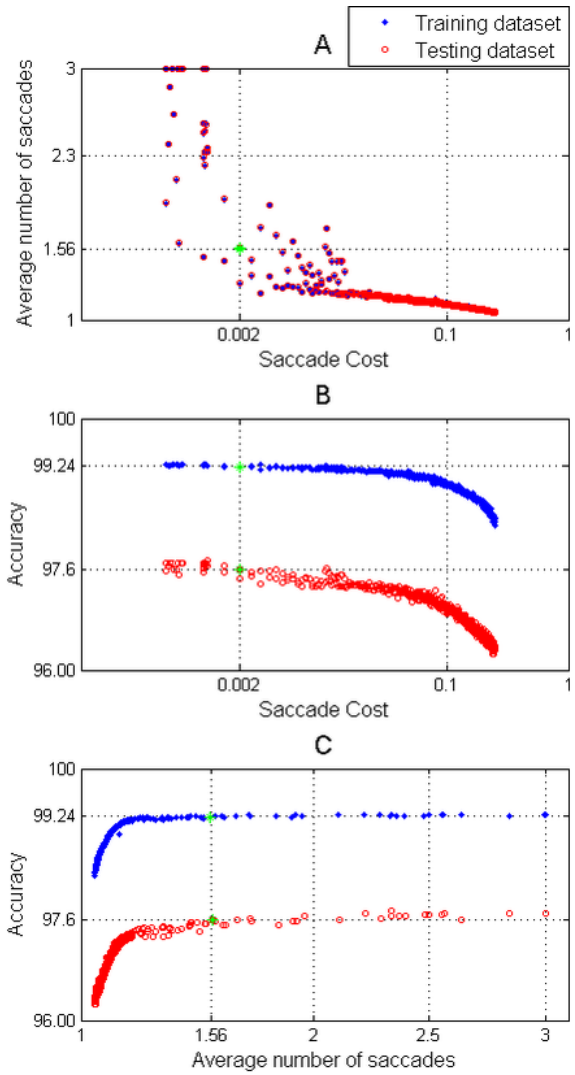


Figure 6-13: Network accuracy for N-MNIST dataset and average number of saccades per sample for different $Cost_{saccade}$. The network was trained with 10 epochs for each $Cost_{saccade}$. The reason that sometimes the plots are not monotonic is because training neural network is a stochastic process and starts from different random states. The average trend is similar to what is expected. Green marks show the accuracy and average number of saccades when $Cost_{saccade}$ is 0.002. In this case, accuracy is close to the highest accuracy of SRNN while rather than using 3 saccades per sample, 1.56 saccades are used in average.

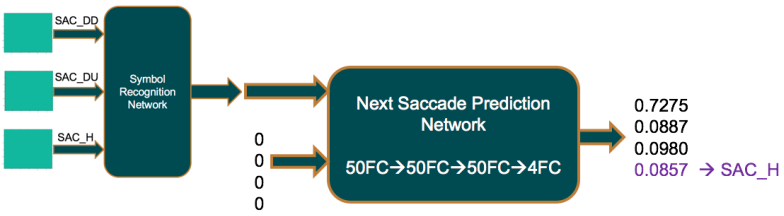


Figure 6-14: The NSPN outputs for the initial saccade

Table 6.8: Second saccade choice from the NSPN for testing samples. 'SAC_N' means no extra saccade was chosen. Error rate here is calculated after performing the second saccade (except for the SAC_N).

Saccade	Percentage of samples	Error rate
SAC_N	48.53%	0.53%
SAC_DD	12.15%	6.17%
SAC_DU	39.32%	4.78%
SAC_H	0.00%	-
Overall	100.00%	2.89%

accuracy versus the average number of saccades for different $Cost_{saccade}$. As the $Cost_{saccade}$ is decreased, the network elicits more saccades and accuracy increases. With a $Cost_{saccade}$ of 0.002 (when the average prediction loss is 0.05 for the training samples), for example, the system needs to perform on average 1.56 saccades per sample to achieve an accuracy of 97.6% for the testing data and 99.2% for the training data. These results are very similar to the results of the NSP block in Section 6.4.2. This experiment shows that if the $Cost_{saccade}$ is adjusted to a reasonable value, the SRNN can maintain its accuracy (see Table 6.7), requiring on average around half of the saccades (1.56 saccades rather than three saccades).

Next we study the features of the NSPN more carefully and provide more results. For the following experiments a $Cost_{saccade}$ of 0.002 is used.

As seen in Fig. 6-14, the NSPN always chooses SAC_H as the initial saccade. Table 6.7 shows that SAC_H is the best saccade, on average, for both training and testing samples.

Table 6.8 shows the NSPN actions after the first saccade. For more than 48% of the samples one single saccade was sufficient, and the error rate for this category was

Table 6.9: Saccade choice statistics from the saccade prediction network for testing samples. 'All SAC+' indicates samples that needed more than three saccades.

Saccade	Percentage of samples	Error rate	Error rate of first saccade
SAC_H	48.53%	0.53%	0.53%
SAC_H→SAC_DD	10.18%	1.37%	9.23%
SAC_H→SAC_DU	36.03%	1.83%	4.30%
All SAC	4.47%	24.16%	51.23%
All SAC+	0.79%	32.91%	60.76%
Overall	100%	2.4%	5.52%

low (0.53%). This means the saccade prediction network correctly determined the samples that were easy to recognize from the first saccade. For the other samples, the network suggested an extra saccade. Since the initial saccade was SAC_H, the NSPN did not recommend SAC_H again for the second saccade, as expected.

After the second saccade, the network may decide that a third saccade is required for some samples. Table 6.9 shows the percentage of the test samples for each combination of saccades after all necessary saccades have been performed. For more than 48% of the samples, the system only asked for one saccade and recognized them with 99.47% accuracy. This means that recognition of these samples was an easy task for the SRNN.

For around 46% of the samples (10.18+36.03), the NSPN asked for one additional saccade. These samples have been recognized with 98.27% accuracy while the same samples have been recognized with 94.61% accuracy before performing the second saccade.

For around 4.5% of the samples, the NSPN asked for two additional saccades. These samples have been recognized with 75.84% accuracy while the same samples have been recognized with only 48.77% accuracy after the first saccade. These samples were hard to recognize and, as expected, the NSPN requested three saccades for them.

When recognition loss is high, the NSPN sometimes asks to perform more than three saccades, repeating one of the previous saccades. In Table 6.9, for 0.79% of the samples, the NSPN requested more than three saccades. The 32.91% error rate in

this group of samples indicates that these samples were very difficult to recognize. In real scenarios (i.e., not a pre-recorded dataset), repeating a saccade may provide additional information.

6.5 Conclusions

Our aim in this Chapter was to answer the question of how to perform saccades with a DVS to improve accuracy, speed and power consumption in a robotic platform. The first step was to mount a DVS on a motorized pan-tilt unit to perform object recognition with saccadic movements. In this step, the objective was to determine the effect of saccade direction, velocity and distance on the information captured by the DVS.

Our experimental results show that to achieve better object recognition the internal parameters of the recognition system should ideally match the saccade velocity while the distance of movement should be sufficiently short. The best saccade direction depends on the shape of the object. The first saccade can be random or move in the direction that, on average, is optimal for all cases. In our experiment, most of the objects could be recognized with the first saccade, although in some cases the system needed to perform an extra saccade to gain enough information for recognition. A proposed analytical approach and later on, an Artificial Neural Network, were used to predict the need for an extra saccade and also predict the best direction for the next saccade based on the information obtained from previous saccades. The schemes were shown to halve the number of saccades required while preserving the accuracy of the network.

Chapter 7

Performance Comparison of Time-Step-Driven versus Event-Driven Neural State Update Approaches in SpiNNaker

This work has been submitted to be publish in:

Amirreza Yousefzadeh, Mikel Soto, Teresa Serrano-Gotarredona, Francesco Galluppi, Luis Plana, Steve Furber, and Bernabe Linares-Barranco, "Performance Comparison of Time-Step-Driven versus Event-Driven Neural State Update Approaches in SpiNNaker", International Symposium on Circuit and System (ISCAS), May2018

7.1 Abstract

The SpiNNaker chip is a multi-core processor optimized for neuromorphic applications. Many SpiNNaker chips are assembled to make a highly parallel million core platform. This system can be used for simulation of a large number of neurons in real-time. SpiNNaker is using a general purpose ARM processor that gives a high amount of flexibility to implement different methods for processing spikes. Various libraries and packages are provided to translate a high-level description of Spiking

Neural Networks (SNN) to low-level machine language that can be used in the ARM processors. In this chapter, we introduce and compare three different methods to implement this intermediate layer of abstraction. We have examined the advantages of each method by various criteria, which can be useful for professional users to choose between them.

7.2 Introduction

The SpiNNaker software sits on top of the SpiNNaker hardware to allow a smooth design and simulation of different neural network configurations. Each SpiNNaker chip runs an application programming interface (API) on top of a specific event-based kernel. The host machine runs a Python package (PyNN) for the specification of the neural network structures.

Using PyNN description language [82], the user can specify different neural network topologies and parameters such as populations, synapses, projections and neuron models. Finally, another specific tool maps the PyNN neural network description to the SpiNNaker resources generating and downloading binary files for real-time simulation of the neural network.

In hardware, a population of neurons is assigned to each ARM core. The states of these neurons are stored in the local data memory and can be updated with different events. ARM cores communicate with each other through a packet switch network on chip. Each packet carries the source address that contains the neuron's ID, core ID, and chip ID. Each chip includes a router that communicates to all the cores and external links. The routing tables of the routers are programmed to establish the predefined neural connections. Additionally, the SDRAM memory in each chip can be used to store synaptic weights and allow each neuron to be connected to a few thousand synapses. Several alternative methods to implement Spiking Neural Networks (SNN) in SpiNNaker (besides the standard PyNN based approach) have been reported to process spikes and store neuron states and synaptic weights [83, 80, 84]. In this work, we present three different methods to map an SNN on SpiNNaker

and compare their performances. All methods use the same neuron model, which is not directly supported in the standard PyNN based approach.

The neuron model used is entirely event-driven. The state of the neuron evolves with time. In our work, we have used signed event-driven spiking neurons, as described in [57]. This neuron is a Leaky Integrated and Fire neuron with two thresholds. When a neuron membrane exceeds the positive threshold, it will generate a positive spike, and when it exceeds a negative threshold, a negative spike will be created. Moreover, leakage in this neuron is linear.

For this work, a previously reported poker card symbol recognition Convolutional Spiking Neural Network (ConvNet) is used as benchmark [57]. This network is composed of 4 convolutional layers interleaved with two subsampling stages. Input spikes come from a Dynamic Vision Sensor [2]. Section 2 describes the different implementations. Section 3 shows the experimental setup and results obtained from each implementation. Finally, Section 4 summarizes the peculiarities of each implementation.

7.3 Alternative Benchmark Implementations on SpiNNaker

7.3.1 Time-Step-Driven Implementation (based on the Standard SpiNNaker Approach)

This implementation was done using the available SpiNNaker software version. A new neuron model compatible with the standard models was created to implement the poker card ConvNet. In standard SpiNNaker software, each neuron will be updated at a regular time step that is 1ms by default.

Our event-driven neuron fires positive and negative spikes, however, the SpiNNaker software does not support negative spikes. Therefore, we have created a new sub-neuron model PN that has two voltage thresholds with the same value but inverse sign. This sub-neuron fires and resets when the membrane voltage exceeds the posi-

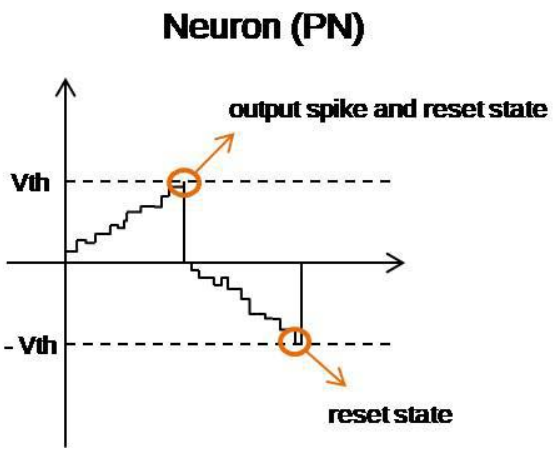


Figure 7-1: Behavior of the neuron that acts as positive neuron (PN)

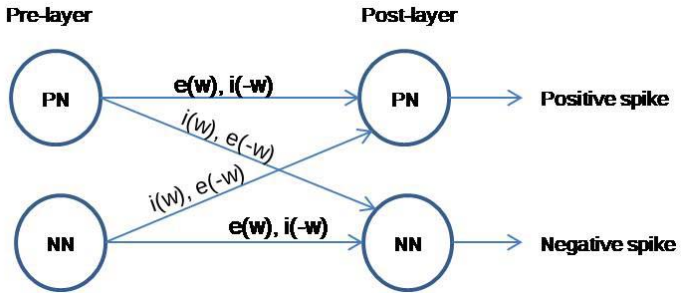


Figure 7-2: Pre-layer post-layer connection scheme. e(w) is excitatory target and i(w) is inhibitory target

tive threshold, and it resets without firing when the membrane voltage goes below the negative threshold, as shown in Fig. 7-1. If we now create a symmetric sub-neuron NN that behaves inversely and combine both {PN, NN}, we get the positive and negative spikes.

The connection between a pre-layer neuron and a post-layer neuron is shown in Fig. 7-2. The connecting weight w can, in principle, be positive or negative. If positive, an excitatory synapse e(w) is used and if negative, an inhibitory synapse i(-w) is used. Positive spikes from pre sub-neuron PN are connected to post sub-neuron PN using {e(w),i(-w)}, but the effect is sign-reversed when connecting to post sub-neuron NN {e(-w),i(w)}. The connections from pre NN to the two post sub-neurons are symmetrically reversed.

The ConvNet architecture to perform recognition of the card symbols of [57] with this implementation needs 130 cores (9 SpiNNaker chips) using 100 neurons per core.

7.3.2 Spike-Driven Implementation

In the original SpiNNaker software, there is a millisecond time step and each neuron will be updated every millisecond. There are two significant disadvantages for this millisecond time step. First, temporal precision will be limited to the time step that is 1ms by default. Second, even without any incoming activity, neurons are continuously updated every time step. If more timing resolution or fully event-driven power consumption is desired, one may prefer a neuron that updates only after receiving a spike. It is believed that most of the information of spikes are at the time of firing [65] and losing timing accuracy may have a significant effect on some SNN implementations.

In this implementation, the original SpiNNaker software has been modified to include the neuron model that generates positive and negative spikes. SpiNNaker uses AER packets [3] for spike communication. To add polarity to spikes, we needed to modify the structure of the AER packets. Additionally, we removed the time step neuron update. In this case, neurons will be updated immediately after receiving spikes.

When a spike enters to a processor, first it will load the proper synaptic weight from SDRAM memory, then it will immediately update the neuron potential. If the spike has the positive polarity, the membrane potential of the neuron is increased by the synaptic weight value. Conversely, if the spike has a negative polarity, the membrane potential will be decreased by the synaptic weight value.

In this implementation there is no time step for every millisecond, so every neuron should keep track of their last time of receiving spike (for calculating leakage) and the last time of generating a spike (for calculating the refractory period). After the update of the membrane potential of the neuron, if it exceeds the positive threshold, a positive post-synaptic spike will be generated, and if it overcomes the negative threshold, a negative post-synaptic spike will be generated.

All the above processes in each core of SpiNNaker chips will be done immediately after a spike enters and it takes around 10us in average. So each core in this implementation can handle around 100k synaptic updates per second independent of how many neurons are inside this core. For this poker card implementation, we used 104 ARM cores in SpiNNaker hardware.

7.3.3 ConvNet Optimized Implementation

The main difference of this ConvNet implementation is that it takes advantage of the weight sharing property of the ConvNets. This weight sharing property highly reduces the number of synaptic weights that must be stored for a neuronal population.

In this implementation, the original SpiNNaker software has been modified to admit a particular "convolution connector" [84]. This "convolution connector" stores in the local data memory of the chip the kernel weights of the corresponding population. Each feature map shares a "convolution connector" for every neuron in it. The implementation is entirely event-driven because each event is processed at the time it arrives. When an event reaches the convolution module, the corresponding kernel is applied to update the neuron state and the neighbor pixels. Being the weights in the local data memory prevents from reading the synaptic weights from the SDRAM.

Furthermore, we distinguish between the shared parameters of the neurons in a population like voltage threshold, leakage rate, refractory time and so on and non-shared parameters like each neuron state and firing times. The shared neuron parameters are stored once per population in the local memory. In the original SpiNNaker software, each neuron parameters are stored in the data memory. Storing the parameters in the local memory provides a high-speed access, but the capacity of this local memory can limit the number of neurons that can be implemented per core. Specifically, we are able to implement 2048 convolution neurons per core, where this number is determined by the maximum number of addressable neurons by the implemented routing scheme. This method uses 22 ARM cores for the poker card recognition benchmark.

7.4 Experimental Setup and Results

The POKER-DVS Database [77] is recorded by shuffling poker cards in front of the DVS. This dataset has high event rate. Processing events and recognizing symbols in real-time is a challenging task. In previous work [57] a convolutional spiking neural network to process poker card symbols is introduced and implemented in a software simulator[1]. In this work, we have implemented this network in PyNN [82] and mapped it to SpiNNaker using the above mentioned three implementation methods.

When we send the spikes in real-time (high event rate) to SpiNNaker, the cores sometimes cannot handle all the spikes with the same speed as they arrive. In this case, the old spikes will be dropped which may decrease the accuracy of the network. A different implementation of the Spiking Neural Network in SpiNNaker can have different processing efficiency and throughput. We measured and compared the accuracy of the previously mentioned methods when presenting the POKER-DVS Database in real-time and with different slowed-down factors to the SpiNNaker boards.

Fig. 7-3 shows one of our experimental setups for real-time experiments with SpiNNaker. We loaded the POKER-DVS events sequence in the Event Player (or data player) board [3]. The AER-Node board [6] receives AER events from the Data-Player and puts them on a fast serial link [69] to be sent to the SPIN-5 board. The processed spikes will be sent back to the AER-Node board on the same fast serial link. Finally, the AER-Node board sends the output spikes to the USB-AER board [3] to be sent to the computer through a USB port.

The classification is considered successful when the number of output events for the correct category is higher than for the other categories. We repeated the experiment for different slowed-down factors of the events of the input stimulus sequence. Also, we applied the same slowed-down factor to the network timing parameters for a correct time scaling. Fig. 7-4 shows the average recognition success rates obtained for different slowed-down factors. A '1' slowed-down factor means real-time operation.

As it can be seen, when the slowed-down rate is very high, all the implementations

[1]This neural network is trained in frame-based domain and converted to SNN by using a proposed algorithm in [57]

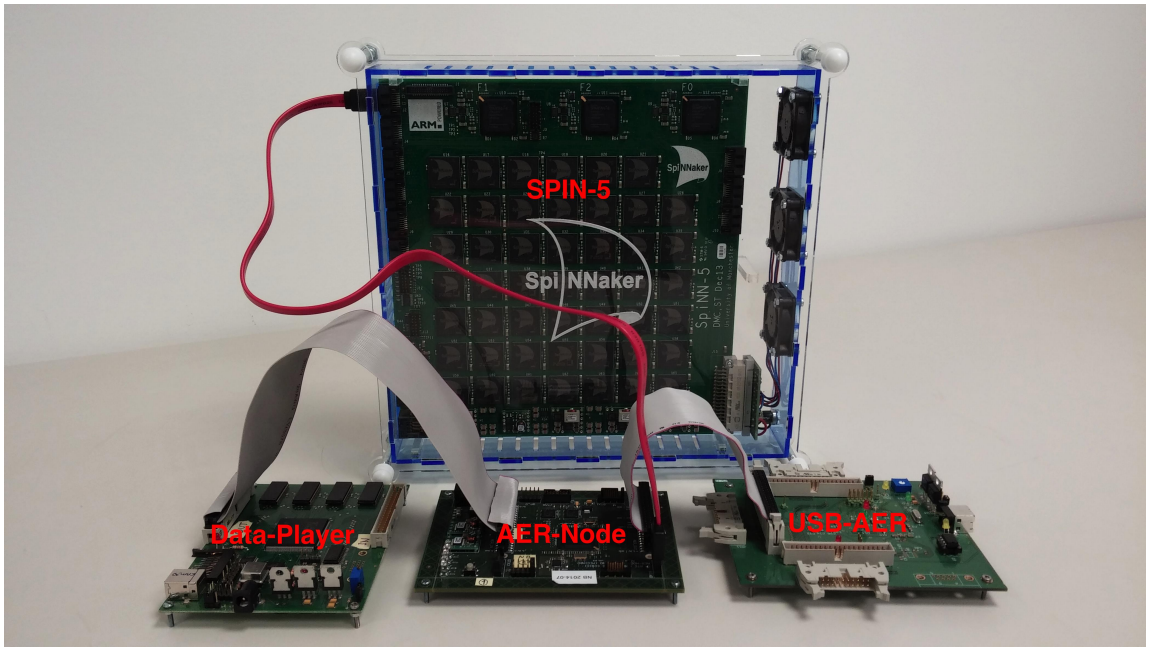


Figure 7-3: Experimental setup. Events flew from Data-Player [3] to AER-NODE board [6] to SPIN-5 board. The processed events come back from the SPIN-5 board to AER-NODE board and they go to USB-AER [3] board to be sent to the computer.

will show almost similar performance, which is close to what is observed when using the software simulator [57]. When spike presentation is closer to real-time, the time-step-driven implementation shows better performance. This result shows that the time-step-driven implementation needs the minimum amount of processing (neuron updates) when the event rate is very high. The spike-driven implementation shows the worst results for high event rates because it needs the maximum processing time per spike among all the methods. Beside processing efficiency, other factors should be considered. For example, while the time-step-driven implementation occupied 130 ARM cores, the ConvNet implementation only needed 22 cores. Additionally, spike timing resolution in the spike-driven implementation is 10us while it is 1ms for the time-step-driven implementation. It is important to mention that the neural network parameters were obtained by mapping from a frame-based training method [57]. Therefore, the network accuracy could be more sensitive to the rate of spikes than the time of spikes.

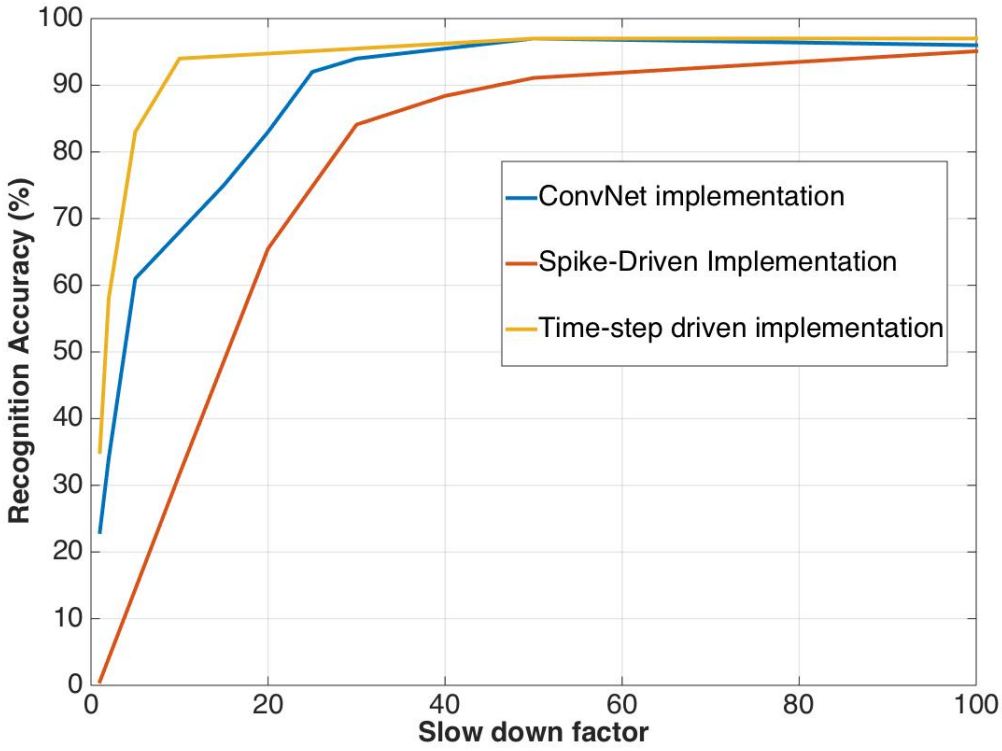


Figure 7-4: Accuracy of symbol recognition versus slow rate of POKER-DVS events from real-time

7.5 Conclusions

In this work, we have compared three different neural network implementations using the SpiNNaker platform. The first implementation is time-step-driven processing which is the standard method for the SpiNNaker software. We have shown that this method has the highest recognition performance and experiences less spike congestion and therefore less event dropping. For the second method, spike-driven implementation, we removed the time step constraint and spikes are processed immediately. We have shown that this method has a high timing resolution in comparison to the time-step-driven method while it needs more processing time per spike. Finally, for the ConvNet optimized implementation, the spikes are processed immediately and the synaptic weights can be stored in the local data memory of the ARM processors rather than in SDRAM. This method has very high timing resolution and good processing performance. When the input event rate is high, this method showed worse performance than the time-step-driven but a better performance than the spike-driven implementation. Also, this method is limited to convolutional connections and cannot be used for fully connected networks.

Chapter 8

Hardware Implementation of Convolutional STDP for On-line Visual Feature Learning

This work has been published in:

A. Yousefzadeh, T. Masquelier, T. Serrano-Gotarredona, and B. Linares-Barranco, "Hardware Implementation of Convolutional STDP for On-line Visual Feature Learning", International Symposium on Circuit and System (ISCAS), May2017

8.1 Abstract

In this chapter, we present a highly hardware friendly STDP (Spike Timing Dependent Plasticity) learning rule for training Spiking Convolutional Cores in Unsupervised mode and training Fully Connected Classifiers in Supervised Mode. Examples are given for a 2-layer Spiking Neural System which learns in real time features from visual scenes obtained with spiking DVS (Dynamic Vision Sensor) Cameras. All the HDL codes are freely available for the academic purpose.

8.2 Introduction

Biological brains continuously learn new incoming information. They never stop learning. Our goal in this chapter is to present an embedded and low power hardware for online unsupervised learning of visual features by using bio-inspired Dynamic Vision Sensors (DVS) [2] and spiking neural networks (SNNs). SNNs have interesting features like event-driven power consumption and pseudo-simultaneity [57].

In this work, we developed and implemented a new algorithm for hardware implementation that has been inspired by Masquelier's pioneering work on STDP (Spike Time Dependent Plasticity) [85]. They developed an algorithm for face recognition using still image frames. Intensity to delay conversion was used to generate artificial spike trains from each frame. They used simple Integrated-and-Fire (IF) neurons without leakage because after each frame presentation, the network resets all neuron states. They allowed at most one spike per each neuron for each frame.

In our application, we wanted to use a DVS camera as the input source. However, a DVS does not use intensity to delay encoding. A DVS pixel generates a signed event when there has been a given relative change in light ($\Delta I/I = C$). Additionally, we wanted to perform training on the continuous input event flow coming from the DVS. For this, we rely on synchrony detection, which is very close to what happens in biological perception [65]. Synchrony based coding is a kind of temporal coding (as opposed to rate coding) but it does not rely on precise spike ordering (as opposed to rank order encoding). What matters is whether or no spikes appear close enough to each other in time. In synchrony-based processing, a visual feature is represented by pseudo-synchronous spikes coming from specific synapses.

Bichler et al. [86] introduced an impressive algorithm for online feature extraction based on STDP. The algorithm successfully detected cars passing from a freeway by using DVS input with unsupervised learning. They used a simplified STDP version to enhance processing speed. However, because they used fully connected neurons topologies, different neurons learned the same features in several positions.

By using Convolutional Neural Networks, one can share a set of weights (kernel)

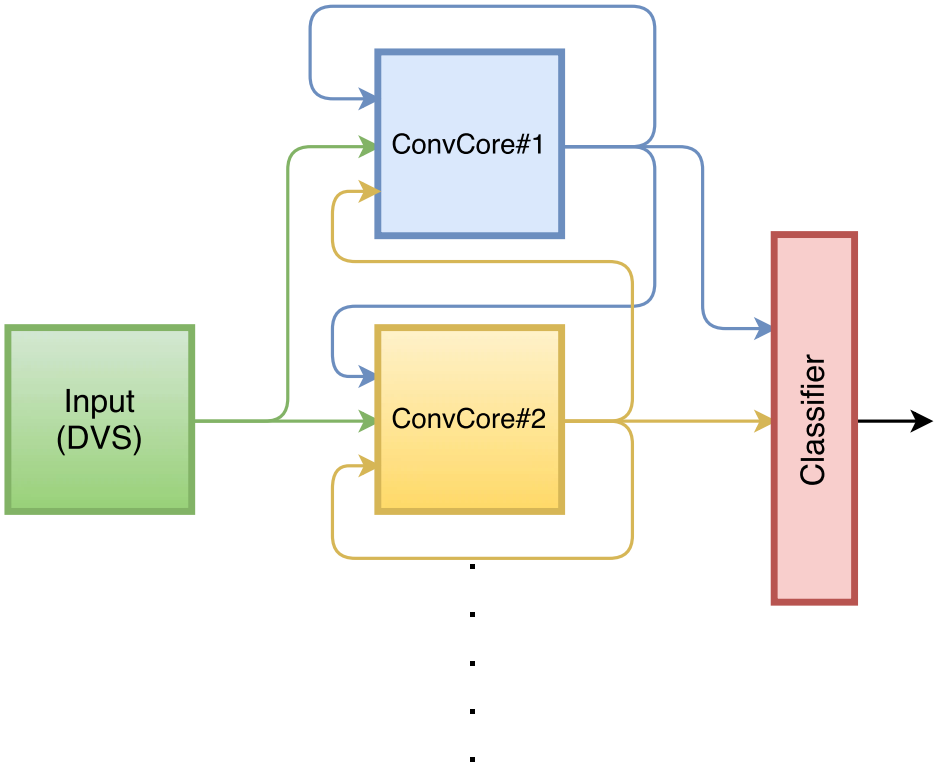


Figure 8-1: Simple Network Topology used in this Work

between many neurons. In this way, each kernel learns a feature independently of its position[1]. Convolutional Spiking Neural Networks (CSNN) are more efficient than fully connected SNNs in terms of processing and memory for pattern recognition tasks.

In the next Sections, we will briefly explain the learning algorithm and its hardware implementation.

8.3 Learning Algorithm

In this work, we used a simple 2-layer neural network, as shown in Fig. 8-1. The first layer is an unsupervised learning convolutional layer for feature extraction. The second layer is a simple supervised-learning classifier. Both layers use a simplified

[1]Chapter 4 explains more about convolutional neural network and it's difference with fully connected one.

STDP-based learning rule, explained later in this Section.

8.3.1 Neuron model

The first layer implements a reduced number of convolutional populations equipped with STDP for feature detection through unsupervised learning. Convolutional populations are called here also "ConvCore". Each ConvCore can have a few kernels that can be fixed or plastic (with STDP). In this Section, we just used one layer of ConvCores, and we used only the positive input spikes coming from the DVS output. For inhibition, we used reset rather than applying an inhibitory kernel. We used simple Leaky Integrate and Fire (LIF) neurons. Each incoming spike adds the synaptic weight to the membrane value [2]. When the neuron's membrane value reaches a threshold, it generates a post-synaptic spike, increases threshold of ConvCore to the current membrane value and reset its membrane value. This simple threshold adaptation mechanism is implemented to regulate the activity of ConvCores and guarantee competition between them.

Leakage in these neurons has been implemented by using an approximation for the exponential decay. One option for leakage implementation is to update each neuron only when it receives spikes. This method is entirely event-driven but needs to keep track of last update time for each neuron. If processing time is more important than memory consumption, such event-driven neuron is recommended. However, in FPGA implementations, memory limitations are typically more stringent. For this reason, we choose to not store the last update time for each neuron, but to apply leakage to all neurons periodically. The update period depends on the stimulus but for standard real-time DVS data, 1ms is reasonable. It will take just a few microseconds in the FPGA to update the leakage for all the neurons[3].

[2]We intentionally use here the term "membrane value", as opposed to the more conventional terminology of membrane voltage or potential, because in our hardware implementation it will be stored as a plain 9-bit integer value in the interval [0, 511], which is entirely different from a physical voltage in the range of millivolts.

[3]Another interesting method to apply leakage is to use spike-based leakage. In this method a constant amount of leakage (in a linear way, exponential or by using bitwise-shift) will be applied to a neuron after receiving a spike. This method is truly event-driven, does not need to keep track of time and is more compatible with our spike-based STDP rule. However we did not use it in this

For the exponential leakage approximation, we implemented the following operation on the membrane value

$$V_{new} = V - (V >> n_{leak}) \tag{8.1}$$

Where symbol $(>> n)$ represents a bitwise right shift of n bits. We used $n_{leak} = 4$, which is equivalent to leaking to 15/16 of the previous membrane value every millisecond. This corresponds to a membrane equivalent time constant of 16ms, which is in the biological range.

The same leakage circuit is used for thresholds as part of threshold adaptation method but with a slower time constant. This leakage will allow decreasing of the threshold for ConvCores that were inactive for a considerable amount of time.

If kernel size is $[K_x, K_y]$, each neuron from the previous layer is connected to $K_x \times K_y$ neurons in a ConvCore with synaptic weights equal to the kernel values. After adding the kernel to membrane values of the neurons in a ConvCore, if a neuron's membrane value exceeds the threshold, an output spike will be generated. Then, the learning process updates the kernel weights.

To guarantee competition during learning, we used a winner-take-all mechanism [87] and inhibitory kernels as shown in Fig. 8-2. First of all, after updating neurons with an event, more than one neuron's membrane value may have reached its threshold. In this case, only the one with the highest value among all the ConvCores will generate a spike. Afterwards, all the neurons in the ConvCore that fired will be reset to their resting value. This will stop neurons in the same ConvCore to learn multiple features. By using this mechanism, a ConvCore can learn only one feature in different positions. At the same time, an inhibitory kernel inhibits neurons in the other ConvCores in the same kernel area (see Fig. 8-2). For this, we used a simple reset rather than applying an inhibitory kernel. This second competition mechanism is needed to discourage different ConvCores from learning the same features.

work and may be investigated in future works.

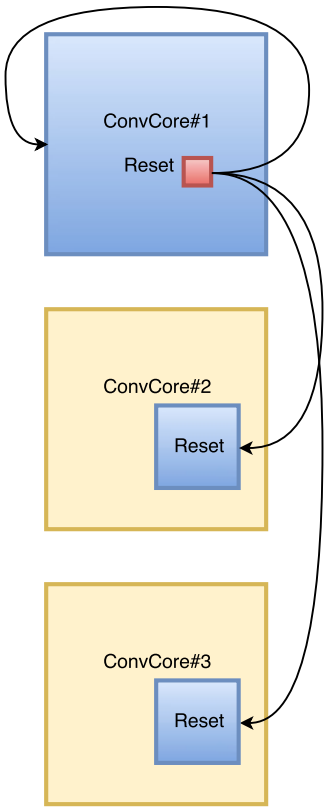


Figure 8-2: Competition mechanisms. After a kernel update in ConvCore#1, the neuron in red has max value after passing the threshold and therefore is the only one spiking within the kernel area. After this, all neurons in ConvCore#1 will be reset, as well as all neurons of the other ConvCores inside the kernel area (region in blue will be reset).

8.3.2 Layer 1: Unsupervised Convolutional STDP Learning

STDP is a bio-inspired learning rule that modifies the strength of a neuron's synapses as a function of the precise temporal relations between pre- and post-synaptic spikes [88]. There are different variants of STDP rules, but all of them share a universal concept. Synaptic weights are updated on a per- spike basis, and the synaptic update depends on the time difference between pre- and post-synaptic spikes.

Here we used a new STDP rule which is highly efficient for hardware implementation, if not the most reported so far. It is very similar to Bichler's proposal [86], where all the synapses of a neuron are equally depressed upon reception of a postsynaptic spike, except for the synapses that were activated with a pre-synaptic spike a short time before, which are potentiated. Therefore, implementation of such a rule needs to store timestamps for the incoming spikes. Also, a buffer is required to save the last incoming spikes, and the size of this buffer depends on the input spike rate. In hardware implementations, usually, buffer sizes are fixed and cannot be adjusted via a parameter. Consequently, it is hard to estimate the best buffer size for all the applications.

We have modified this rule to make it more hardware friendly. In the proposed STDP rule rather than limiting the pre- to post- time window, we defined the number of synapses to be potentiated. This way, when a postsynaptic spike is generated, a logic block will find a specific number of active synapses that have contributed in the firing. In our proposed rule, there is no need to do time stamping on spikes because always a predefined number of synapses will be potentiated. If parameters are chosen carefully, leakage will not allow a neuron to fire in case the last pre-synaptic spikes arrived a long time before the postsynaptic spike, thus preserving synchrony. This rule also stops general potentiation or general depression. Also, we added another mechanism to equally potentiate all the selected synapses regardless of the number of spike coming from a synapse. Kernel weights are normalized after potentiation. This way, all the synapses will be depressed equally with an adaptive rate.

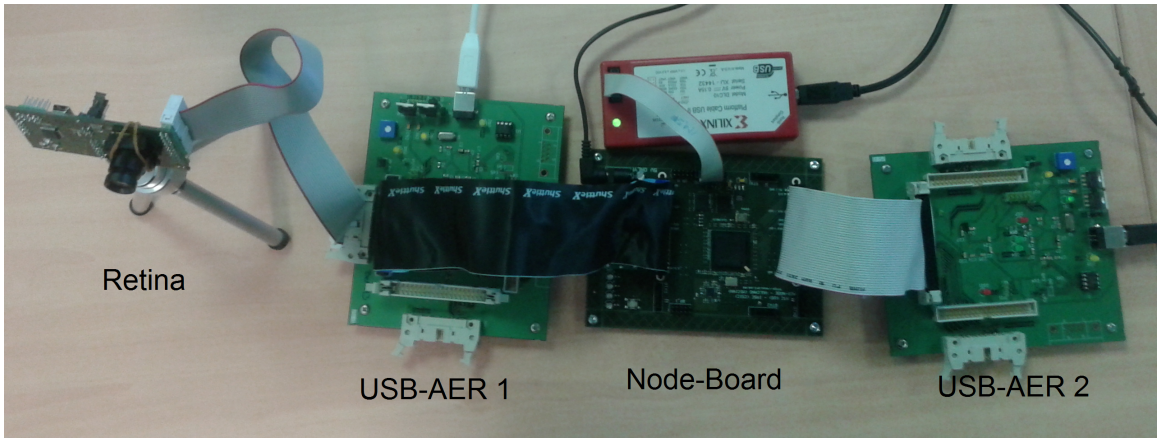


Figure 8-3: Hardware Setup for Learning Experiments

8.3.3 Layer 2: Supervised STDP Learning

To classify the ConvCores output activities, a layer of fully connected supervised STDP neurons has been designed for output layer of our spiking neural network. A supervised STDP neuron has an external input (called "supervisor"), also encoded through AER (Address Event Representation), which forces post-synaptic spikes from this neuron when its representative "category" (or feature) is present at the input. Therefore, whenever a "supervisor" spike arrives, the corresponding active synapses will be potentiated. Otherwise, active synapses will be depressed.

8.4 Hardware Implementation

To do real-time learning and feature extraction, we implemented the above algorithm with HDL (Hardware Description Language) on FPGA. Fig. 8-3 shows the hardware setup that was used. We used a silicon retina (DVS) as input and a Spartan-6 FPGA Node-Board [6] for the network. USB-AER 2 [3] boards were used to send spikes in real-time to a computer for visualization.

Fig. 8-4 shows the block diagram of the FPGA implementation inside the Node-Board Spartan-6 FPGA. It contains ConvCores and supervised STDP Neurons core and AER interfaces (to handle asynchronous communications outside FPGA). The number of ConvCores and the configuration of layers can be customized. Different

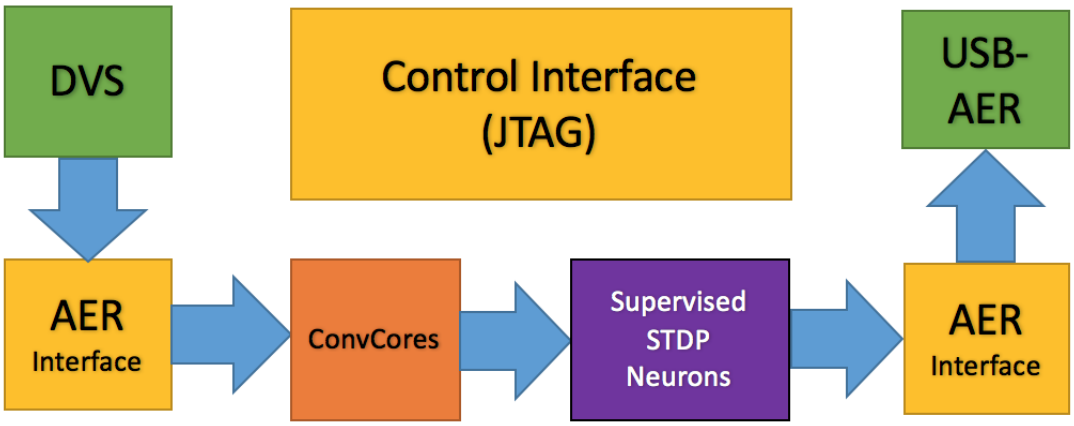


Figure 8-4: FPGA System Implementation Block Diagram (DVS and USB-AER boards are outside of FPGA)

cores communicate with Address Event Representation (AER) events [3].

A conceptual block diagram of the ConvCores is shown in Fig. 8-5. Each ConvCore contains a convolutional processor to perform convolution and two RAMs to keep the neurons membrane values (Neuron RAM) and synaptic weights (Kernel RAM). A STDP processor is shared between all ConvCores in one block because STDP events are rare and only one STDP processor is fast enough to handle them. The STDP processor is connected to a circular buffer to keep the last spikes and use them in STDP learning. Xilinx's ChipScope debug tool is used to program the initial parameters and to monitor the kernels evolution online using a computer.

The amount of resources needed to implement the ConvCore scheme on FPGAs depends on the number of neurons and kernels. For example, once we implemented this core on a Spartan-6 FPGA (XC6SLX150T-3) with 32×32 input pixels, six kernels with 9×9 weights and 512 words for the circular buffer. With these parameters, ConvCores take 1276 slices (out of 23K available slices) of FPGA. Among these occupied slices, 587 slices belong to convolutional processors, and 537 slices belong to STDP processor. Update of membrane values for each input event takes 90 clock cycles, and STDP learning takes less than 900 clock cycles for updating kernels. Also, for each millisecond, one leakage update process takes 1025 clock cycles. When clock

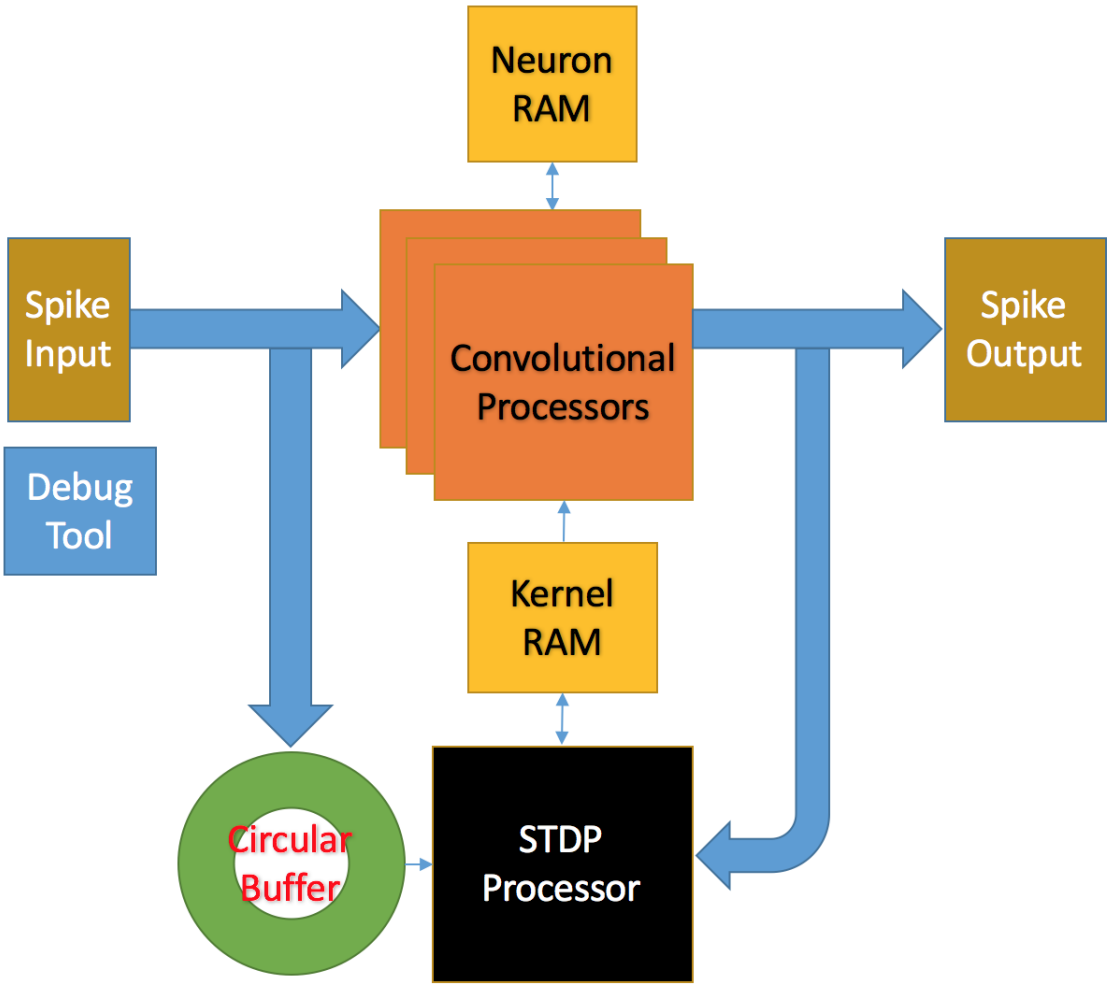


Figure 8-5: Simplified Block diagram of ConvCore in FPGA

frequency is 100MHz, each convolution takes 0.9us, and each STDP process takes less than 9us. These delays are reasonable for online learning in real-time.

8.5 Implementation Results

To test our network, we used two simple letters ('A' and 'B') and moved them in front of the DVS to generate spikes. Fig. 8-6 (a) shows a screenshot from the jAER software [4] used to visualize DVS spikes. The classifier can be trained online along with STDP layer or afterwards.

STDP kernels learn the features that repeat more. For Convolutional STDP, when kernel size is larger than the object, we expect the kernel to learn the whole object. Otherwise, kernels should learn just some parts of the objects as features. Different objects may have shared features, so it is natural to extract characteristic features and use them for object recognition.

We tested our hardware with kernel sizes of 9×9 while sub-sampling the output of the DVS from 128×128 down to 32×32. First, we used two ConvCores. Fig. 8-6 (b) shows the reconstruction of kernel weights after learning the input shown in Fig. 8-6 (a). In this case, because sizes of objects are smaller than 9×9, kernels learned the whole object templates.

In another experiment, we presented the same stimulus to DVS but closer, so that the objects resulted in sizes larger than the kernels, as shown in Fig. 8-6 (c). In this case, we used eight kernels of 7×7. The kernels learned characteristic features from the two letters. Reconstructed kernel weights for this experiment are shown in Fig. 8-6 (d).

8.6 Conclusion

In this chapter, we have introduced a digital implementation of an algorithm for online STDP learning of visual features by using real live visual data from a DVS camera. A video demonstration of the whole chip including parameter adjustment

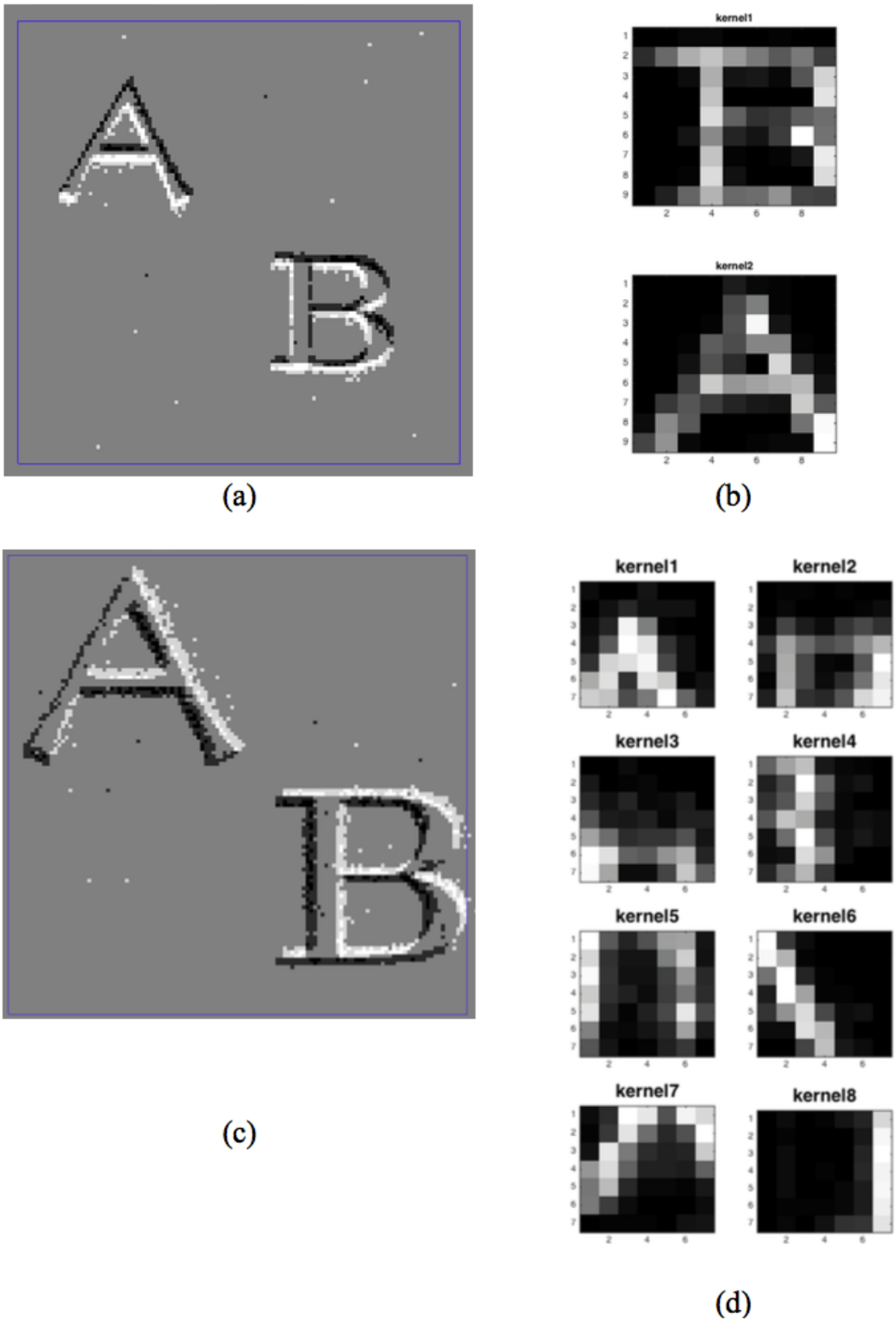


Figure 8-6: (a, c) Screen captures of jAER software to visualize DVS output. Black dots show negative spikes while white dots show positive ones. (b, d) Reconstruction of kernel weights after learning. To see the complete recording videos (including parameters) and online evolution of kernels refer to [8] [9].

through ChipScope, online STDP learning and online learning of classifier is provided in [89].

Chapter 9

List of Publications and Patents

9.1 Journal Papers

1. A. Yousefzadeh, L. Plana, S. Temple, T. Serrano-Gotarredona, S. Furber, and B. Linares-Barranco, "Fast Predictive Handshaking in Synchronous FPGAs for Fully Asynchronous Multi-Symbol Chip Links. Application to SpiNNaker 2-of-7 Links," IEEE Transactions on Circuits and Systems, Part-II.

2. A. Yousefzadeh, M. Jablonski, T. Iakymchuk, A. Linares-Barranco, A. Rosado, L. Plana, T. Serrano-Gotarredona, S. Furber, and B. Linares-Barranco, "On Bit-Serial AER links for Multi- FPGA Spiking Neural Hardware with Multiple Links per FPGA Compatible with Fully Asynchronous Parallel AER Links," IEEE Transactions on Biomedical Circuits and Systems.

3. A. Yousefzadeh, G. Orchard, T. Serrano-Gotarredona, B. Linares-Barranco, "Active Perception with Dynamic Vision Sensors", IEEE Transactions on Biomedical Circuits and Systems, Under review.

4. E. Stromatias, A. Yousefzadeh, M. Soto, T. Serrano-Gotarredona, B. Linares-Barranco, "Stochastic Spike-Timing Dependent Plasticity With Binary Weights", Frontiers in Neuroscience, under review.

9.2 Conference Papers

1. A. Yousefzadeh, T. Serrano-Gotarredona, and B. Linares-Barranco, "Fast Pipeline 128x128 Pixel Spiking Convolution Core for Event-Driven Vision Processing in FPGAs," IEEE Int. Event Based Conference on Control and Signal Processing, Krakow, Poland, June 2015.

2. A. Yousefzadeh, M. Jablonski, T. Iakymchuk, A. Linares-Barranco, A. Rosado, L. Plana, T. Serrano-Gotarredona, S. Furber, and B. Linares-Barranco, "Multiplexing AER Asynchronous Channels over LVDS Links with Flow-Control and Clock- Correction for Scalable Neuromorphic Systems", ISCAS2017

3. A. Yousefzadeh, M. Jablonski, T. Iakymchuk, A. Linares-Barranco, A. Rosado, L. Plana, T. Serrano-Gotarredona, S. Furber, and B. Linares-Barranco, "Live Demonstration: Multiplexing AER Asynchronous Channels over LVDS Links with Flow-Control and Clock- Correction for Scalable Neuromorphic Systems", ISCAS2017

4. A. Yousefzadeh, T. Masquelier, T. Serrano-Gotarredona and B. Linares-Barranco, "Hardware Implementation of Convolutional STDP for on-line visual feature extraction", ISCAS2017

5. A. Yousefzadeh, T. Masquelier, T. Serrano-Gotarredona and B. Linares-Barranco, "Live Demonstration: Hardware Implementation of Convolutional STDP for on-line visual feature extraction", ISCAS2017

6. S. Hoseini, G. Orchard, A. Yousefzadeh, B. Deverakonda, T. Serrano-Gotarredona and B. Linares-Barranco, "Passive localization and detection of quadcopter UAVs by using Dynamic Vision Sensor", 5th Iranian joint conference on fuzzy and intelligent systems

7. S.J Thorpe, A. Yousefzadeh, J. Martin and T. Masquelier, "Unsupervised learning of repeating patterns using a novel STDP based algorithm", Vision Science Society, May 2017

8. A. Yousefzadeh, M. Soto, T. Serrano-Gotarredona, B. Linares-Branco, "Performance Comparison of Time-Step-Driven versus Event-Driven Neural State Update Approaches in SpiNNaker", ISCAS2018, under review.

9. S. Hoseini, G. Orchard, A. Yousefzadeh, B. Deverakonda, T. Serrano-Gotarredona and B. Linares-Barranco, "Hardware implementation of Real-Time Temporal Frequency Detection with Event-Based Vision Sensor", ISCAS2018, under review

10. A. Yousefzadeh, G. Orchard, E. Stromatias, T. Serrano-Gotarredona, B. Linares-Barranco, "Hybrid-NN, An Efficient Low-Power Digital Hardware Implementation for Event-based Artificial Neural Networks", ISCAS2018, under review.

9.3 Patents

1. Unsupervised detection of repeating patterns in a series of events, European Patent Office EP16306525 Nov-2016, Amirreza Yousefzadeh, Timothee Masquelier, Jacob Martin, Simon Thorpe, Licensed to the Californian start-up BrainChip.

2. Method, digital electronic circuit and system for unsupervised detection of repeating patterns in a series of events, European Patent Office EP17305186 Feb-2017, Amirreza Yousefzadeh, Bernabe Linares-Barranco, Timothee Masquelier, Jacob Martin, Simon Thorpe, Exclusive Licensed to the Californian start-up BrainChip.

3. Method and Apparatus for Stochastic STDP with Binary Weights, U.S. Patent and Trademark office 012055.0440P Nov-2017, Evangelos Stromatias, Amirreza Yousefzadeh, Teresa Serrano-Gotarredona, Bernabe Linares-Barranco, Licensed to Samsung Advanced Institute of Technology.

Chapter 10

References

[1] S. B. Furber, D. R. Lester, L. A. Plana, J. D. Garside, E. Painkras, S. Temple, and A. D. Brown, "Overview of the spinnaker system architecture," *IEEE Transactions on Computers*, vol. 62, pp. 2454–2467, Dec 2013.

[2] T. Serrano-Gotarredona and B. Linares-Barranco, "A 128 × 128 1.5% contrast sensitivity 0.9% *FPN* 3us latency 4 mw asynchronous frame-free dynamic vision sensor using transimpedance preamplifiers," *IEEE Journal of Solid-State Circuits*, vol. 48, pp. 827–838, March 2013.

[3] R. Serrano-Gotarredona, M. Oster, P. Lichtsteiner, A. Linares-Barranco, R. Paz-Vicente, F. Gomez-Rodriguez, L. Camunas-Mesa, R. Berner, M. Rivas-Perez, T. Delbruck, S. C. Liu, R. Douglas, P. Hafliger, G. Jimenez-Moreno, A. C. Ballcels, T. Serrano-Gotarredona, A. J. Acosta-Jimenez, and B. Linares-Barranco, "Caviar: A 45k neuron, 5m synapse, 12g connects/s aer hardware sensory-processing-learning-actuating system for high-speed visual object recognition and tracking," *IEEE Transactions on Neural Networks*, vol. 20, pp. 1417–1438, Sept 2009.

[4] F.Corradi, S. Bamford, L. Longinotti, and T. Delbruck, "jaer software." `https://sourceforge.net/projects/jaer`.

[5] C. Posch, D. Matolin, and R. Wohlgenannt, "A qvga 143 db dynamic range frame-free pwm image sensor with lossless pixel-level video compression and time-domain cds," *IEEE Journal of Solid-State Circuits*, vol. 46, pp. 259–275, Jan 2011.

[6] T. Iakymchuk, A. Rosado, T. Serrano-Gotarredona, B. Linares-Barranco, A. Jiménez-Fernández, A. Linares-Barranco, and G. Jiménez-Moreno, "An aer handshake-less modular infrastructure pcb with x8 2.5gbps lvds serial links," in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1556–1559, June 2014.

[7] A. Yousefzadeh, "Real time demo, spiking MNIST recognition using DVS." `https://youtu.be/FRqH7kRaBW8`, 2016.

[8] A. Yousefzadeh, "Real-time video, online unsupervised visual feature extraction." `https://youtu.be/05NyI3qp05g`.

[9] A. Yousefzadeh, "Real-time video, live demonstration: On-line visual feature learning in FPGA." `https://youtu.be/VRz1uLXa4KA`.

[10] L. A. Plana, S. B. Furber, S. Temple, M. Khan, Y. Shi, J. Wu, and S. Yang, "A gals infrastructure for a massively parallel multiprocessor," *IEEE Design Test of Computers*, vol. 24, pp. 454–463, Sept 2007.

[11] L. A. Plana, J. Bainbridge, S. Furber, S. Salisbury, Y. Shi, and J. Wu, "An on-chip and inter-chip communications network for the spinnaker massively-parallel neural net simulator," in *Second ACM/IEEE International Symposium on Networks-on-Chip (nocs 2008)*, pp. 215–216, April 2008.

[12] S. Furber and A. Brown, "Biologically-inspired massively-parallel architectures - computing beyond a million processors," in *2009 Ninth International Conference on Application of Concurrency to System Design*, pp. 3–12, July 2009.

[13] K. A. Boahen, "Point-to-point connectivity between neuromorphic chips using address events," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 47, no. 5, pp. 416–434, 2000.

[14] C. Posch, T. Serrano-Gotarredona, B. Linares-Barranco, and T. Delbruck, "Retinomorphic event-based vision sensors: Bioinspired cameras with spiking output," *Proceedings of the IEEE*, vol. 102, pp. 1470–1484, Oct 2014.

[15] B. Son, Y. Suh, S. Kim, H. Jung, J. S. Kim, C. Shin, K. Park, K. Lee, J. Park, J. Woo, Y. Roh, H. Lee, Y. Wang, I. Ovsiannikov, and H. Ryu, "4.1 a 640x480 dynamic vision sensor with a 9um pixel and 300meps address-event representation," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 66–67, Feb 2017.

[16] M. A. Sivilotti, *Wiring considerations in analog VLSI systems, with application to field-programmable networks*. PhD thesis, California Institute of Technology, 1991.

[17] F. Stefanini, S. Sheik, E. Neftci, and G. Indiveri, "Pyncs: a microkernel for high-level definition and configuration of neuromorphic electronic systems," *Frontiers in Neuroinformatics*, vol. 8, no. 73, 2014.

[18] X. X. Lagorce, S.-H. Ieng, X. Clady, M. Pfeiffer, and R. B. Benosman, "Spatiotemporal features for asynchronous event-based data," *Frontiers in Neuroinformatics*, vol. 9, no. 46, 2015.

[19] G. Orchard, C. Meyer, R. Etienne-Cummings, C. Posch, N. Thakor, and R. Benosman, "Hfirst: A temporal approach to object recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 37, pp. 2028–2040, Oct 2015.

[20] J. Conradt, F. Galluppi, and T. C. Stewart, "Trainable sensorimotor mapping in a neuromorphic robot," *Robotics and Autonomous Systems*, vol. 71, no. Sup-

plement C, pp. 60 – 68, 2015. Emerging Spatial Competences: From Machine Perception to Sensorimotor Intelligence.

[21] I. E. Sutherland, "Micropipelines," *Commun. ACM*, vol. 32, pp. 720–738, June 1989.

[22] Z. Zhang, *Performance analysis of synchronization circuits*. PhD thesis, School Comput. Sci., Univ. Manchester, 2010.

[23] Y. Li, B. Nelson, and M. Wirthlin, "Synchronization techniques for crossing multiple clock domains in FPGA-based *TMR* circuits," *IEEE Transactions on Nuclear Science*, vol. 57, pp. 3506–3514, Dec 2010.

[24] M. Mahowald, "VLSI Analogs of Neuronal Visual Processing: a Synthesis of Form and Function," *PhD, Computation and Neural Systems, Caltech, Pasadena, California*, 1992.

[25] V. Chan, C. Jin, and A. van Schaik, "An Address-Event Vision Sensor for Multiple Transient Object Detection," *IEEE Trans. on Biomedical Circuits and Systems*, pp. 278–288, Dec. 2007.

[26] Z. Fu, T. Delbrück, P. Lichsteiner, and E. Culurciello, "An Address-Event Fall Detector for Assisted Living Applications," *IEEE Trans. on Biomedical Circuits and Systems*, pp. 88–96, June 2008.

[27] B. Wen and K. Boahen, "A Silicon Cochlea with Active Coupling," *IEEE Trans. on Biomedical Circuits and Systems*, pp. 444–455, Dec. 2009.

[28] S. Mitra, S. Fusi, and G. Indiveri, "Real-time classification of complex patterns using spike-based learning in neuromorphic vlsi," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 3, pp. 32–42, Feb 2009.

[29] R. Mill, S. Sheik, G. Indiveri, and S. L. Denham, "A Model of Stimulus-Specific Adaptation in Neuromorphic Analog VLSI," *IEEE Trans. on Biomedical Circuits and Systems*, pp. 413–419, Oct. 2011.

[30] D. G. Chen, D. Matolin, A. Bermak, and C. Posch, "Pulse-Modulation Imaging – Review and Performance Analysis," *IEEE Trans. on Biomedical Circuits and Systems*, pp. 64–82, Feb. 2011.

[31] M. M. Khan, D. R. Lester, L. A. Plana, A. Rast, X. Jin, E. Painkras, and S. B. Furber, "Spinnaker: Mapping neural networks onto a massively-parallel chip multiprocessor," pp. 2849–2856, June 2008.

[32] J. Fieres, J. Schemmel, and K. Meier, "Realizing Biological Spiking Network Models in a Configurable Wafer-Scale Hardware System," *IEEE Int. Joint. Conf. on Neural Networks (IJCNN-WCCI)*, pp. 969–976, June 2008.

[33] P. O. Pouliquen and A. G. Andreou, "Bit-Serial Address-Event Representation," *Proceedings of 33rd Annual Conference on Information Sciences and Systems, Baltimore MD, USA*, March 1999.

[34] K. Boahen, "A Burst-Mode Word-Serial Address-Event Link I,II,III," *IEEE Trans. Circ. and Syst. Part I*, vol. 51, no. 7, pp. 1269–1300, July 2004.

[35] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana, "The spinnaker project," *Proceedings of the IEEE*, vol. 102, pp. 652–665, May 2014.

[36] N. Semiconductors, "LVDS owner's manual," *4th-Edition*, 2008.

[37] L. Miro-Amarante, A. Jimenez, A. Linares-Barranco, F. Gomez-Rodriguez, R. Paz, G. Jimenez, A. Civit, and R. Serrano-Gotarredona, "A lvds serial aer link," in *2006 13th IEEE International Conference on Electronics, Circuits and Systems*, pp. 938–941, Dec 2006.

[38] H. Berge and P. Häfliger, "High-Speed Serial AER on FPGA," *Proc. IEEE Int. Symp. on Circ. and Syst. (ISCAS)*, pp. 857–860, May 2007.

[39] D. B. Fasnacht, A. M. Whatley, and G. Indiveri, "A Serial Communication Infrastructure for Multi-Chip Address Event Systems," *Proc. IEEE Int. Symp. on Circ. and Syst. (ISCAS)*, pp. 648–651, May 2008.

[40] C. Zamarreño-Ramos, A. Linares-Barranco, T. Serrano-Gotarredona, and B. Linares-Barranco, "Multi-Casting Mesh AER: A Scalable Assembly Approach for Reconfigurable Neuromorphic Structured AER Systems. Application to ConvNets," *IEEE Trans. Biomedical Circ. and Syst.*, vol. 7, pp. 82–102, Feb. 2013.

[41] C. Zamarreño-Ramos, T. Serrano-Gotarredona, and B. Linares-Barranco, "A 0.35um Sub-ns Wake-up Time ON-OFF Switchable LVDS Driver-Receiver Chip I/O Pad Pair for Rate-Dependent Power Saving in AER Bit-Serial Links," *IEEE Trans. on Biomedical Circuits and Systems*, vol. 6, pp. 486–497, October 2012.

[42] C. Zamarreño-Ramos, R. Kulkarni, J. Silva-Martínez, T. Serrano-Gotarredona, and B. Linares-Barranco, "A 1.5ns OFF/ON Switching-Time Voltage-Mode LVDS Driver/Receiver Pair for Asynchronous AER Bit-Serial Chip Grid Links with up to 40 Times Event-Rate Dependent Power Savings," *IEEE Trans. on Biomedical Circuits and Systems*, vol. 7, pp. 722–731, October 2013.

[43] C. Zamarreño-Ramos, T. Serrano-Gotarredona, and B. Linares-Barranco, "An Instant-Startup Jitter-Tolerant Manchester-Encoding Serializer/Deserializar Scheme for Event-Driven Bit-Serial LVDS Inter-Chip AER Links," *IEEE Trans. Circ. and Syst., Part I*, vol. 58, no. 11, pp. 2647–2660, 2011.

[44] L. Plana, J. Heathcote, J. Pepper, S. Davidson, J. Garside, S. Temple, and S. Furber, "spi/o: A library of FPGA designs and reusable modules for i/o in spinnaker systems." `http://dx.doi.org/10.5281/zenodo.51476,2014`.

[45] P. Popescu, A. Solheim, and M. Wight, "Experimental Monolithic High Speed Transceiver for Manchester Encoded Data," *Proc. of the 1995 Bipolar/CMOS Circ. and Tech. Meeting*, pp. 110–113, Oct. 1995.

[46] P. A. Franaszek and A. X. Widmer, "Byte Oriented DC Balanced (0,4) 8b/10b Partitioned Block Transmission Code," *US Patent 4,486,738*, Dec. 4, 1984.

[47] M. Jabłoński, T. Serrano-Gotarredona, and B. Linares-Barranco, "High-speed serial interfaces for event-driven neuromorphic systems," in *2015 International*

Conference on Event-based Control, Communication, and Signal Processing (EBCCSP), pp. 1–4, June 2015.

[48] R. Dobkin and R. Ginosar, "Zero Latency Synchronizers Using Four and Two Phase Protocols," *VLSI Systems Research Center, Technion - Israel Inst. of Techn. Internal Report*, October 2007.

[49] R. Ginosar, "Fourteen ways to fool your synchronizer," in *Ninth International Symposium on Asynchronous Circuits and Systems, 2003. Proceedings.*, pp. 89–96, May 2003.

[50] Xilinx, "Spartan-6 FPGA GTP transceivers (advance spec),"

[51] L. Camunas-Mesa, A. Acosta-Jimenez, C. Zamarreno-Ramos, T. Serrano-Gotarredona, and B. Linares-Barranco, "A 32×32 pixel convolution processor chip for address event vision sensors with 155 ns event latency and 20 meps throughput," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 58, pp. 777–790, April 2011.

[52] L. Camunas-Mesa, C. Zamarreno-Ramos, A. Linares-Barranco, A. J. Acosta-Jimenez, T. Serrano-Gotarredona, and B. Linares-Barranco, "An event-driven multi-kernel convolution processor module for event-driven vision sensors," *IEEE Journal of Solid-State Circuits*, vol. 47, pp. 504–517, Feb 2012.

[53] A. Linares-Barranco, R. Paz-Vicente, F. Gómez-Rodriguez, A. Jiménez, M. Rivas, G. Jiménez, and A. Civit, "On the *AER* convolution processors for FPGA," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pp. 4237–4240, May 2010.

[54] T. Delbruck, "Frame-free dynamic digital vision," in *in Proc. Int. Symp.Secure-Life Electron., Adv. Electron. Quality Life Soc*, pp. 21–26, March 2008.

[55] R. Serrano-Gotarredona, T. Serrano-Gotarredona, A. Acosta-Jimenez, and B. Linares-Barranco, "A neuromorphic cortical-layer microchip for spike-based

event processing vision systems," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 53, pp. 2548–2566, Dec 2006.

[56] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, November 1998.

[57] J. A. Perez-Carrasco, B. Zhao, C. Serrano, B. Acha, T. Serrano-Gotarredona, S. Chen, and B. Linares-Barranco, "Mapping from frame-driven to frame-free event-driven vision systems by low-rate rate coding and coincidence processing–application to feedforward convnets," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, pp. 2706–2719, Nov 2013.

[58] Xilinx, "Spartan-6 *FPGA* block *RAM* resources," *UG383*.

[59] G. K. Cohen, G. Orchard, S. H. Leng, J. Tapson, R. Benosman, and A. van Schaik, "Skimming digits: Neuromorphic classification of spike-encoded images," *Frontiers in Neuroscience*, 2016.

[60] G. Orchard, A. Jayawan, G. K. Cohen, and N. Thakor, "Converting static image datasets to spiking neuromorphic datasets using saccades," *Frontiers in Neuroscience*, vol. 9, p. 437, 2015.

[61] S. R. Kheradpisheh, M. Ganjtabesh, S. J. Thorpe, and T. Masquelier, "STDP-based spiking deep neural networks for object recognition," *CoRR*, vol. abs/1611.01421, 2016.

[62] J. H. Lee, T. Delbruck, and M. Pfeiffer, "Training deep spiking neural networks using backpropagation," *Frontiers in Neuroscience*, 2016.

[63] P. U. Diehl, D. Neil, J. Binas, M. Cook, S. C. Liu, and M. Pfeiffer, "Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing," in *2015 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, July 2015.

[64] R. V. Rullen and S. J. Thorpe, "Rate coding versus temporal order coding: What the retinal ganglion cells tell the visual cortex," *Neural Computation*, vol. 13, pp. 1255–1283, June 2001.

[65] T. Masquelier, G. Portelli, and P. Kornprobst, "Microsaccades enable efficient synchrony-based coding in the retina: a simulation study," *Scientific Reports*, vol. 6, 2016.

[66] A. Luczak, B. L. McNaughton, and K. D. Harris, "Packet-based communication in the cortex," *Nature Reviews Neuroscience*, vol. 16, no. 12, 2015.

[67] A. G. Andreou, A. A. Dykman, K. D. Fischl, G. Garreau, D. R. Mendat, G. Orchard, A. S. Cassidy, P. Merolla, J. Arthur, R. Alvarez-Icaza, B. L. Jackson, and D. S. Modha, "Real-time sensory information processing using the truenorth neurosynaptic system," in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 2911–2911, May 2016.

[68] A. Amir, B. Taba, D. Berg, T. Melano, J. McKinstry, C. Di Nolfo, T. Nayak, A. Andreopoulos, G. Garreau, M. Mendoza, J. Kusnitz, M. Debole, S. Esser, T. Delbruck, M. Flickner, and D. Modha, "A low power, fully event-based gesture recognition system," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[69] A. Yousefzadeh, M. Jabłoński, T. Iakymchuk, A. Linares-Barranco, A. Rosado, L. A. Plana, T. Serrano-Gotarredona, S. Furber, and B. Linares-Barranco, "On multiple aer handshaking channels over high-speed bit-serial bidirectional lvds links with flow-control and clock-correction on commercial FPGAs for scalable neuromorphic systems," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 11, pp. 1133–1147, Oct 2017.

[70] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)* (J. Fürnkranz and T. Joachims, eds.), pp. 807–814, Omnipress, 2010.

[71] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[72] "FLASH-MNIST-DVS dataset." `http://www2.imse-cnm.csic.es/caviar/MNISTDVS.html`.

[73] A. Yousefzadeh, T. Masquelier, T. Serrano-Gotarredona, and B. Linares-Barranco, "Hardware implementation of convolutional stdp for on-line visual feature learning," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–4, May 2017.

[74] A. Yousefzadeh, "Real time demo, spiking neural network in FPGA for handwritten digit recognition." `https://youtu.be/U-mrRcaQkpI`, 2017.

[75] R. Engbert, "Microsaccades: a microcosm for research on oculomotor control, attention, and visual perception," in *Visual PerceptionFundamentals of Vision: Low and Mid-Level Processes in Perception* (S. Martinez-Conde, S. Macknik, L. Martinez, J.-M. Alonso, and P. Tse, eds.), vol. 154, Part A of *Progress in Brain Research*, pp. 177 – 192, Elsevier, 2006.

[76] A. Bruckstein, R. J. Holt, I. Katsman, and E. Rivlin, "Head movements for depth perception: Praying mantis versus pigeon," *Autonomous Robots*, vol. 18, no. 1, pp. 21–42, 2005.

[77] T. Serrano-Gotarredona and B. Linares-Barranco, "Poker-DVS and MNIST-DVS. their history, how they were made, and other details," *Frontiers in Neuroscience*, 2015.

[78] P. Diehl and M. Cook, "Unsupervised learning of digit recognition using spike-timing-dependent plasticity," *Frontiers in Computational Neuroscience*, vol. 9, p. 99, 2015.

[79] X. Lagorce, G. Orchard, F. Gallupi, B. E. Shi, and R. Benosman, "Hots: A hierarchy of event-based time-surfaces for pattern recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PP, no. 99, pp. 1–1, 2016.

[80] G. Orchard, X. Lagorce, C. Posch, S. B. Furber, R. Benosman, and F. Galluppi, "Real-time event-driven spiking neural network object recognition on the spinnaker platform," in *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 2413–2416, May 2015.

[81] A. Yousefzadeh, "Real time demo, symbol recognition and saccade prediction network." `https://youtu.be/Tw5Jbi_on-4`, 2016.

[82] A. Davison, D. Bruderle, J. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger, "Pynn: a common interface for neuronal network simulators," *Frontiers in Neuroinformatics*, vol. 2, p. 11, 2009.

[83] X. Lagorce, E. Stromatias, F. Galluppi, L. A. Plana, S.-C. Liu, S. B. Furber, and R. B. Benosman, "Breaking the millisecond barrier on spinnaker: implementing asynchronous event-based plastic models with microsecond resolution," *Frontiers in Neuroscience*, vol. 9, p. 206, 2015.

[84] T. Serrano-Gotarredona, B. Linares-Barranco, F. Galluppi, L. Plana, and S. Furber, "ConvNets experiments on SpiNNaker," *Proceedings - IEEE International Symposium on Circuits and Systems*, vol. 2015-July, pp. 2405–2408, 2015.

[85] T. Masquelier and S. J. Thorpe, "Unsupervised learning of visual features through spike timing dependent plasticit," *PLoS Comput Biol*, 2007.

[86] O. Bichler, D. Querlioz, S. J. Thorpe, J.-P. Bourgoin, and C. Gamrat, "Extraction of temporally correlated features from dynamic vision sensors with spike-timing-

dependent plasticity," *Neural Networks*, vol. 32, no. Supplement C, pp. 339 – 348, 2012. Selected Papers from IJCNN 2011.

[87] M. Oster, R. Douglas, and S.-C. Liu, "Computation with spikes in a winner-take-all network," *Neural Computation*, vol. 21, no. 9, pp. 2437–2465, 2009. PMID: 19548795.

[88] H. Markram, J. Lübke, M. Frotscher, and B. Sakmann, "Regulation of synaptic efficacy by coincidence of postsynaptic aps and epsps," *Science*, vol. 275, no. 5297, pp. 213–215, 1997.

[89] A. Yousefzadeh, "Real-time video, convolutional *STDP* and supervised *STDP* neurons in FPGA real-time learning." `https://youtu.be/_J7s_xLLBbA`.